UNIVERSITÉ DE MONTRÉAL

MODÉLISATION DES ACCÈS MÉMOIRE LORS DE LA MULTIPLICATION D'UNE MATRICE CREUSE PAR UN VECTEUR SUR PROCESSEUR GRAPHIQUE

THALIE LÉNA KEKLIKIAN

DÉPARTEMENT DE GÉNIE ÉLECTRIQUE

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION

DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES

(GÉNIE ÉLECTRIQUE)

DÉCEMBRE 2014

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

MODÉLISATION DES ACCÈS MÉMOIRE LORS DE LA MULTIPLICATION D'UNE MATRICE CREUSE PAR UN VECTEUR SUR PROCESSEUR GRAPHIQUE

présenté par : KEKLIKIAN Thalie Léna

en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de :

M. DAVID Jean Pierre, Ph. D., président

M. SAVARIA Yvon, Ph. D., membre et directeur de recherche

M. LANGLOIS J.M. Pierre, Ph. D., membre et codirecteur de recherche

M. BOYER François-Raymond, Ph. D., membre

REMERCIEMENTS

J’aimerais prendre l’opportunité, ici, de remercier Yvon Savaria, mon directeur, qui m’a encouragé à poursuivre mes études aux cycles supérieurs en me proposant un projet qui convenait parfaitement à mes goûts. Il m’a soutenu tout au long de ma maîtrise, respectant mes choix sur le cheminement du projet et m’accordant de son temps bien qu’il n’en ait pas beaucoup. Je lui suis reconnaissante de son approche plus globale au projet qui m’a permis de toujours me remettre sur le bon chemin et d’avoir une vision du futur.

Je souhaite aussi remercier mon codirecteur Pierre Langlois pour le soutien que j’ai pu recevoir de lui dans mes moments les plus difficiles. Il a su m’accorder énormément de temps pour répondre à mes questions et réviser mes travaux, m’encourageant toujours en me faisant voir le bon côté des choses. Sa disponibilité m’a permis de ne jamais me décourager à poursuivre mes projets jusqu’au bout. Je lui serai toujours reconnaissante de m’avoir trouvé une opportunité de travail dans mon domaine qui m’a permis de confirmer que c’est le domaine dans lequel je veux travailler.

J’aimerais remercier Pierre Popovic qui a été l’instigateur de mon choix de poursuivre mes études et celui qui m’a donné l’idée originale de mon projet. Il m’a toujours accordé une énorme confiance, me donnant toujours des défis qui m’ont permis de m’affirmer davantage et d’avoir une meilleure estime de moi-même.

J’aimerais aussi remercier ma famille qui me soutient aveuglément, mes amies et la gang des braves qui m’ont fait penser à autre chose pendant ce long parcours et les étudiants du GRM, passés et présents, qui ont su mettre de la vie dans le local et à l’école. Je mentionne particulièrement Sylvain Charasse, Anaïs Saint-Jean, Adrien Larbanet, Keven Chaussé, François Rajotte, Lloyd Salvant et Marc-Aurèle Charpentier-Podrez.

RÉSUMÉ

Une matrice creuse est une matrice dont une grande proportion des éléments sont nuls. Il est souvent avantageux de ne pas tenir compte des éléments nuls lors d'une opération utilisant une matrice creuse. Plusieurs formats de représentation réduisent l'espace nécessaire pour représenter une matrice creuse. Dans la résolution de systèmes linéaires creux, la matrice creuse est souvent multipliée par un vecteur dense. Cette opération est fréquemment utilisée dans des domaines comme l'électromagnétique, le classement de pages web ou des problèmes de transport et de livraison.

Depuis quelques années, les processeurs graphiques (GPU) deviennent de plus en plus performants et s'intègrent aux superordinateurs. La multiplication entre une matrice creuse et un vecteur (SpMV – *Sparse Matrix-Vector Multiplication*) peut être implémentée sur un GPU, puisque l'opération peut se diviser en une série de produits scalaires indépendants. Par contre, c'est une opération difficile à optimiser. Plusieurs travaux ont été réalisés sur son accélération et les auteurs s'entendent pour dire que son exécution est ralentie par les accès à la mémoire irréguliers et difficiles à prévoir.

Dans ce mémoire, nous présentons un modèle permettant de calculer le nombre d'accès à la mémoire de la SpMV implémentée sur GPU pour quatre formats de représentation : le format de compression de ligne (CSR), le format ELLPACK (ELL), le format de coordonnées (COO) et un format hybride entre ELL et COO abrégé à HYB. Le modèle permet de prédire la performance d'une SpMV selon la matrice utilisée, son format et le GPU sur lequel elle est implémentée. Une SpMV ayant moins d'accès à la mémoire aura une meilleure performance. Le modèle calcule le nombre de requêtes à la mémoire et le nombre de transactions à la mémoire. Une requête représente une demande de lecture ou d'écriture et une transaction représente le nombre de transferts de données déclenchés par une requête.

Pour valider le modèle, des implémentations de la SpMV ont été exécutées sur deux cartes GPU récentes, la NVIDIA GeForce GTX 670 et la NVIDIA Tesla K20c. Les matrices utilisées pour les tests sont de tailles différentes et de structures irrégulières. Les résultats montrent que dans le meilleur des cas, le modèle estime parfaitement le nombre de transactions ou de requêtes. Les moyennes des erreurs entre les nombres de transactions estimés et mesurés pour deux implémentations du format CSR, le format ELL, le format COO et le format HYB sont de 1.1 %, 2.6 %, 0 %, 0.7 % et 0.3 % respectivement. De plus, le meilleur format pour la grande majorité des matrices testées est le format HYB, puisqu'il a requis le moins de transactions et de requêtes.

ABSTRACT

A sparse matrix is a matrix in which the majority of elements are zeroes. It is almost always beneficial to avoid calculations on the zero elements when using sparse matrices. For this purpose, several formats have been created to compress sparse matrices by storing only the nonzero elements. An important operation involving a sparse matrix is the sparse matrix-vector multiplication (SpMV) that can be found in many sparse linear solvers. The SpMV is used in many high performance computing applications in fields such as electromagnetics, ranking of web pages or transportation problems.

Graphics processing units have become more powerful over the past few years and are now integrated to supercomputers. The SpMV can be implemented on a GPU since the operation can be divided in multiple dot products between a row of the matrix and the vector. However, the SpMV is difficult to optimize and its acceleration has been studied by several authors. It mostly suffers from poor memory access performance which is difficult to anticipate.

In this project, we propose a model predicting the number of requests and transactions to the memory of the SpMV implemented on GPU. The four formats that were explored are the compressed sparse row format (CSR), the ELLPACK format (ELL), the coordinated format (COO) and a format that is a hybrid between ELL and COO called HYB. Our model can help predict the performance of a SpMV depending on the sparse matrix, the format and the GPU used. When a memory access is requested, it can cause one or multiple data transactions depending of the transfer size. A SpMV implementation has a better performance when the number of memory transactions that it requires is less than with another one.

Our model is validated on two recent GPU boards: the NVIDIA GeForce GTX 670 and the NVIDIA Tesla K20c. Sparse matrices used are chosen from different fields and are unstructured so that the results apply to a larger range of matrices. The results show that, in the best case scenario, the estimated number of transactions and requests exactly match the profiled ones. With respect to the number of memory transactions, the average errors between the estimated numbers and the profiled ones are 1.1 %, 2.6 %, 0 %, 0.7 % and 0.3 % for two implementation of the CSR format, the ELL format, the COO format and the HYB format respectively. Also, the HYB format shows the best performance, since it requires the smallest number of memory transactions and requests.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

BCOO	Block Coordinate
BSR	Block Compressed Sparse Row
COO	Coordinate
CPU	Central Processing Unit
CSC	Compressed Sparse Column
CSR	Compressed Sparse Row
CUDA	Compute Unified Device Architecture
DMA	Direct Memory Access
ECC	Error-correcting code
ELL	ELLPACK
FLOPS	Floating-point Operations per Second
FPGA	Field-Programmable Gate Array
GPGPU	General-Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
HYB	Format de matrice creuse hybride entre ELLPACK et COO
JAD	Jagged Diagonal
JDS	Jagged Diagonal Storage
PCIe	Peripheral Component Interconnect Express
SpMV	Sparse Matrix-Vector Multipication

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

1.1 Contexte

Une matrice creuse est une matrice dont les éléments sont majoritairement nuls. Si une matrice est de taille $n \times n$ et que le nombre d'éléments non nuls qu'elle contient est de l'ordre de n ou moins, on peut la qualifier de creuse. Lorsqu'une matrice est creuse, il peut être plus avantageux d'exécuter des opérations seulement sur ses éléments non nuls. Les matrices creuses sont souvent utilisées dans la description de phénomènes naturels. Par exemple, une matrice représentant la communication entre différents employés d'une grande entreprise serait creuse si chaque employé ne communique qu'avec une petite fraction de tous les autres employés de la société [1]. Un autre exemple est le cas d'un problème de transport où la matrice serait remplie par les frais de transport d'une source i à une destination j. Dans ce cas, le problème consiste à minimiser les frais totaux de transport avec une résolution de problème par recherche du vecteur propre [2]. Dans un autre cas important, l'algorithme PageRank [3] utilisé par Google pour classer les pages web en ordre de priorité utilise une méthode itérative permettant de trouver le vecteur propre d'un système d'interconnexions entre différentes pages web. On retrouve aussi les matrices creuses dans des domaines comme la géodésie et la photogrammétrie [4] ainsi que dans l'analyse des circuits électriques et électroniques [5].

Les opérations élémentaires de l'algèbre de matrices creuses comprennent la transposition, la permutation de lignes ou de colonnes, l'addition et la multiplication de deux matrices, et la multiplication d'une matrice creuse et d'un vecteur [6]. L'opération de multiplication d'une matrice creuse par un vecteur (SpMV – *Sparse Matrix-Vector multiplication*) est utilisée dans un grand nombre de méthodes itératives des systèmes linéaires [7] telles que la méthode d'élimination de Gauss et de Gauss-Jordan ou la recherche des vecteurs et des valeurs propres. Souvent, la matrice utilisée sera de très grande taille et l'exécution d'une telle opération peut consommer beaucoup de temps. Ainsi, l'accélération de la SpMV a été étudiée par plusieurs.

Les processeurs graphiques (GPU – *Graphics Processing Units*) sont des processeurs vectoriels de type SIMD (*Single Instruction Multiple Data*); ils effectuent un même calcul sur différentes données. Ce sont des processeurs très performants et peu coûteux. Selon Owens et al. [8], les GPU

deviennent de plus en plus performants, et ce, plus rapidement que les processeurs à calculs génériques (CPU – *Central Processing Units*), grâce à une plus grande utilisation des transistors dont ils se composent pour du calcul arithmétique. Les CPU, optimisés pour effectuer du code séquentiel, utilisent plutôt une grande partie de leurs transistors pour des techniques de prédiction de branchement et de parallélisme d'instruction. Les processeurs graphiques sont conçus afin de répondre à une classe d'applications parallélisables nécessitant une importante intensité de calcul et où le débit est plus important que la latence [9]. Leurs milliers de cœurs permettent de maximiser le débit de traitement d'une application. La première fonction d'un GPU fut l'affichage graphique, mais plusieurs autres applications peuvent aussi bénéficier du parallélisme qu'ils offrent comme les opérations vectorielles. Compte tenu de ses différents avantages de calculs, les GPU sont de plus en plus utilisés pour exécuter du calcul générique. On appelle cette programmation GPGPU (*General-Purpose Graphics Processing Unit*).

1.2 Motivation

L'opération de multiplication d'une matrice creuse par un vecteur est importante dans plusieurs domaines de l'ingénierie, mais elle est difficile à optimiser. En effet, à cause de la nature creuse de la matrice, les accès aux éléments non nuls sont généralement irréguliers. De plus, cela peut faire en sorte qu'il y ait peu de réutilisation des éléments du vecteur, puisque chaque ligne n'a pas souvent des éléments non nuls aux mêmes colonnes. Pour tenter d'accélérer cette opération, plusieurs auteurs ont proposé des formats de stockage pour matrice creuse permettant de régler les problèmes d'accès non consécutifs. S'ensuit une série d'optimisations possibles selon le domaine d'où la matrice provient, la plateforme utilisée, l'implémentation de l'algorithme, etc. Il est alors nécessaire de trouver un moyen facile de prédire la performance d'une SpMV selon tous ces facteurs afin de faire le meilleur choix possible.

Il est possible de paralléliser la SpMV en divisant l'opération en une série de produits scalaires indépendants entre chaque ligne de la matrice et le vecteur. Grâce à leur architecture de vectorisation où chaque unité logique et arithmétique effectue la même opération, les GPU sont un bon choix pour effectuer des opérations sur des matrices et des vecteurs. Pour effectuer une SpMV, tous les cœurs du GPU pourraient multiplier un élément de la matrice avec un élément du vecteur au même moment. Contrairement aux CPU, les GPU ont beaucoup plus de cœurs réduisant ainsi le nombre d'itérations qu'il faudra réaliser afin de multiplier chaque élément. De plus, le temps

d'accès à la mémoire des GPU est plus rapide que pour les CPU. Par rapport aux FPGA où il est aussi possible de créer plusieurs multiplieurs en parallèle, l'architecture des GPU est déjà disponible et prête à être utilisée. Par contre, les GPU ont aussi certains désavantages. Par exemple, ils présentent d'importants ralentissements lorsqu'une instruction ne peut pas être effectuée sur tous les fils d'exécution en même temps suite à une condition de branchement. De plus, les tailles de leurs mémoires et de leurs caches sont plus petites en comparaison de celles que l'on retrouve sur les CPU conventionnels. Toutefois, cela renforce encore plus le besoin d'un modèle de performance permettant de choisir les optimisations qui éviteraient ces désavantages.

1.3 Objectifs de recherche

Pour prédire la performance de l'implémentation de la SpMV, nous avons besoin d'un modèle. La SpMV est une opération qui, notamment sur un GPU, peut nécessiter plus de cycles consacrés aux instructions d'accès en mémoire que de cycles consacrés au calcul. Nous posons alors l'hypothèse que la performance de l'algorithme peut être estimée d'abord grâce au nombre de transactions à la mémoire. Afin de tester cette hypothèse, nous avons choisi quatre formats de stockage pour les matrices creuses qui permettent d'implémenter la SpMV de cinq façons différentes. Un modèle sera proposé pour calculer le nombre de requêtes et de transactions à la mémoire d'une matrice en particulier, selon l'implémentation choisie et le processeur graphique sur lequel l'algorithme sera exécuté. Une requête représente une demande de lecture ou d'écriture à la mémoire et une transaction représente un transfert de la mémoire vers les registres.

Afin de vérifier notre hypothèse, les cinq implémentations seront réalisées sur GPU et profilées. Les matrices choisies pour les tests sont non structurées afin de toucher à un plus grand ensemble d'applications. Nous utilisons des processeurs graphiques récents afin d'observer le comportement de la SpMV sur cette nouvelle architecture, mais le modèle peut être appliqué à n'importe qu'elle architecture la précédant. Enfin, les nombres de requêtes et de transactions estimés à l'aide du modèle seront comparés à des mesures de profilage.

Le modèle proposé dans ce mémoire permettra de prédire la performance de la multiplication entre une matrice creuse et un vecteur sur GPU à l'aide du nombre de transactions à la mémoire. Il permettra de mieux comprendre la façon dont une SpMV est implémentée sur un GPU et de comparer la performance de ces implémentations selon la matrice testée.

1.4 Plan du mémoire

Le mémoire est divisé en cinq chapitres. Le Chapitre 2 traite des matrices creuses et des processeurs graphiques. Nous y verrons différentes représentations de matrices creuses dans la mémoire, une présentation de la multiplication entre une matrice creuse et un vecteur et les applications qui l'exploitent, une introduction sur les processeurs graphiques et une revue de littérature sur l'accélération de la SpMV sous différentes formes. Ensuite, le Chapitre 3 présentera le modèle proposé. Nous y verrons la méthodologie utilisée et un modèle différent pour chaque format et implémentation choisis. Le Chapitre 4 présentera les résultats obtenus après l'expérimentation. Nous comparerons d'abord les résultats du modèle avec les mesures du profilage et discuterons de la performance du modèle, de l'estimation du temps d'exécution et de la différence de performance entre les formats. Finalement, le mémoire conclura au Chapitre 5 sur l'ensemble des contributions, les améliorations possibles et les travaux futurs.

CHAPITRE 2 MATRICES CREUSES ET PROCESSEURS GRAPHIQUES

Dans ce chapitre, nous allons discuter des éléments entourant la multiplication d'une matrice creuse et d'un vecteur sur processeur graphique. Le chapitre est séparé en cinq sections. Nous introduirons d'abord les matrices creuses, leurs caractéristiques et leurs utilisations dans la première section. S'ensuivra une présentation des différentes représentations des matrices creuses dans la mémoire; les avantages et désavantages de chacune de celles-ci. Dans la troisième section, nous discuterons de l'algorithme choisi dans ce mémoire qui est la multiplication entre une matrice creuse et un vecteur et présenterons un exemple d'application. La section suivante présentera ce qu'il faut savoir sur les processeurs graphiques dans le contexte de ce mémoire. Enfin, la section cinq permettra d'exposer une revue de littérature sur le sujet de la multiplication d'une matrice creuse et d'un vecteur ainsi que sur la modélisation d'algorithme sur des processeurs graphiques.

2.1 Matrices creuses

Une matrice creuse est une matrice ayant une majorité d'éléments nuls. Barrett et al. [7] considèrent une matrice comme creuse lorsqu'il devient plus avantageux pour un algorithme d'éviter de réaliser son opération sur les éléments nuls de la matrice. Selon Tewarson [1] une matrice de taille $n \times n$ est qualifiée de creuse si elle contient un nombre d'éléments non nuls de l'ordre de n.

Les matrices creuses sont souvent décrites à l'aide de la fraction des éléments nuls qu'elle contient, appelée vacuité (*sparsity*). À l'inverse, la fraction des éléments non nuls est appelée densité. Les paramètres décrits au Tableau 2.1 serviront à décrire les matrices creuses.

Tableau 2.1 : Paramètres utilisés pour décrire une matrice creuse

Paramètre	**Description**
K	Nombre de colonnes dans les matrices du format ELLPACK (section 2.2.4)
M	Nombre de lignes de la matrice
maxRow	Nombre maximal d'éléments d'une ligne de la matrice
N	Nombre de colonnes de la matrice
N_{NZ}	Nombre d'éléments non nuls de la matrice
Srow	Tableau répertoriant la taille de chaque ligne
Sp	Vacuité de la matrice creuse (*sparsity*)
d	Densité de la matrice creuse

La vacuité et la densité d'une matrice creuse sont calculées par le rapport du nombre d'éléments nuls ou non nuls sur le nombre d'éléments de la matrice (2.1) et (2.2).

$$Sp = \frac{MN - N_{NZ}}{MN} \tag{2.1}$$

$$d = \frac{N_{NZ}}{MN} \tag{2.2}$$

2.2 Représentation des matrices creuses

Lorsque la taille d'une matrice creuse devient importante, il est plus avantageux de ne stocker que les éléments non nuls de la matrice. Les formats de stockages sont originaires des bibliothèques de résolution de systèmes linéaires creux, dont ITPACK [10], NSPCG [11] ou SPARSKIT [12] tous les trois en langage FORTRAN. Nous comparons ici les avantages et les désavantages de l'utilisation de certains d'entre eux en mettant l'emphase sur ceux qui sont disponibles dans les bibliothèques pouvant être utilisées avec CUDA, soit cuSPARSE [13] et CUSP [14].

2.2.1 Format de coordonnées

Le format le plus simple est le format de coordonnées (COO – *Coordinate*) qui répertorie l'emplacement et la valeur de chaque élément non nul de la matrice. Trois vecteurs sont utilisés : un vecteur pour les indices de ligne, un vecteur pour les indices de colonne et un vecteur pour les valeurs non nulles. La taille de ces vecteurs est égale au nombre d'éléments non nuls de la matrice, que nous dénoterons N_{NZ}. Par exemple, pour un stockage en ordre de lignes, la matrice A de la Figure 2.1 est représentée par les vecteurs *row*, *column* et *value*.

Ce format est notamment utilisé afin de partager des matrices creuses. Par exemple, la banque de matrices creuses de l'université de Floride [15] permet d'importer les matrices s'y trouvant en trois formats distincts. Une matrice creuse de type .mat pouvant servir sur MATLAB, un format nommé Rutherford/Boeing orienté pour le langage FORTRAN et un format nommé Matrix Market [16] où les données sont inscrites en utilisant le format de coordonnées.

Le format Matrix Market permet de stocker des matrices creuses ou denses. Elles peuvent être composées de nombres réels, complexes, entiers ou booléens. Enfin, les éléments de la matrice

peuvent être énumérés de façon générale, où tous les éléments sont présents, ou de façon symétrique, où seule la moitié des éléments est listée. À l'exception des matrices symétriques, une simple lecture du fichier correspond au format COO de la matrice. Des routines de lecture et d'écriture de ce format en C et en MATLAB sont disponibles sur le site internet de la société Matrix Market [17].

$$A = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 10 & 0 & 12 & 0 \\ 0 & 21 & 0 & 0 \\ 0 & 0 & 32 & 0 \end{bmatrix}$$

$$row = [0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 2 \quad 3] \quad column = [1 \quad 2 \quad 3 \quad 0 \quad 2 \quad 1 \quad 2]$$

$$value = [1 \quad 2 \quad 3 \quad 10 \quad 12 \quad 21 \quad 32]$$

Figure 2.1 : Exemple de matrice stockée en format COO

L'avantage de ce format est qu'il est facile à créer, l'accès à la mémoire est systématique et la logique à utiliser est simple. Le désavantage est qu'il y a de la redondance, notamment dans les vecteurs *row* ou *column*. La parallélisation d'une opération sur une matrice stockée en format COO peut se faire en divisant le travail sur chaque élément non nul. Cette division est facile puisque chaque fil d'exécution parallèle peut s'occuper d'un indice du vecteur de valeurs et utiliser le même pour les vecteurs de lignes et de colonnes. Ce genre de parallélisation répond bien à des exemples d'addition de matrices ou de multiplication avec un scalaire. Par contre, il est plus difficile de réaliser la multiplication d'une matrice par un vecteur puisque la division du travail se fait par ligne de la matrice. Si la matrice creuse est stockée en format COO, il est plus difficile, pour les fils d'exécutions parallèles, de connaître le début et la fin d'une même ligne.

2.2.2 Formats de coordonnées compressées

Compression de ligne

Pour remédier à la redondance du format COO, deux formats sont disponibles : un format de compression de ligne (CSR – *Compressed Sparse Row*) et un format de compression de colonne (CSC – *Compressed Sparse Column*). Pour le format CSR, l'idée est de modifier le vecteur *row* en le remplaçant par un vecteur de pointeur de ligne *ptr*. Ainsi, chaque élément du vecteur représente l'indice de début d'une ligne, permettant de n'avoir qu'à stocker $M + 1$ éléments pour ce vecteur.

Le format CSC est présenté de la même façon, mais les éléments non nuls du vecteur de valeurs sont placés en ordre de colonne (*column-major*). Ainsi, le format a besoin du vecteur d'indice de ligne *row* et le vecteur *ptr* représente les indices des éléments commençant chaque colonne de la matrice *A*.

$$A = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 10 & 0 & 12 & 0 \\ 0 & 21 & 0 & 0 \\ 0 & 0 & 32 & 0 \end{bmatrix}$$

$$ptr = [0 \quad 3 \quad 5 \quad 6 \quad 7] \; column = [1 \quad 2 \quad 3 \quad 0 \quad 2 \quad 1 \quad 2]$$

$$value = [1 \quad 2 \quad 3 \quad 10 \quad 12 \quad 21 \quad 32]$$

Figure 2.2 : Exemple de matrice stockée en format CSR

Ce format est très utilisé pour des accélérations de multiplication entre une matrice creuse et un vecteur à cause de la facilité de paralléliser le travail entre chaque ligne de la matrice. En effet, il est possible de connaître à l'avance, le nombre d'éléments non nuls par ligne et l'emplacement de ceux-ci dans le vecteur *value*. Le format CSR est utilisé par plusieurs auteurs qui seront présentés à la section 2.5.

Compression de ligne par bloc

Une autre variante est le format de compression de ligne par bloc (BSR – *Block Compressed Sparse Row*) qui est utilisé lorsque la matrice ciblée peut être décomposée en plusieurs sous-matrices. Prenons la matrice *B* suivante qui peut se diviser en quatre sous-matrices.

$$B = \left[\begin{array}{cc|cc} 0 & 1 & 2 & 3 \\ 10 & 0 & 12 & 0 \\ \hline 0 & 0 & 22 & 0 \\ 0 & 0 & 32 & 0 \end{array}\right] \equiv C = \begin{bmatrix} I & J \\ 0 & K \end{bmatrix}$$

$$ptr = [0 \quad 2 \quad 3] \; column = [0 \quad 1 \quad 1]$$

$$value = [I \quad J \quad K] = [0 \quad 1 \quad 10 \quad 0 \quad 2 \quad 3 \quad 12 \quad 0 \quad 22 \quad 0 \quad 32 \quad 0]$$

Figure 2.3 : Exemple de matrice stockée en format BSR

Le vecteur *value* liste chaque sous-matrice en ordre de ligne, le vecteur *column* indique la colonne à laquelle chaque sous-matrice est placée dans la nouvelle matrice C et le vecteur *ptr* indique les

indices de ligne de chaque première sous-matrice qui se trouve sur une nouvelle ligne. De plus, deux paramètres indiquent le nombre de lignes et de colonnes d'un bloc.

Ce format ne serait pas avantageux pour la matrice *A* utilisée dans la Figure 2.2. Si nous divisons la matrice *A* en quatre sous-matrices, chacune d'entre elles contient un élément non nul. Ceci fait en sorte que le vecteur *value* devra énumérer tous les éléments de la matrice *A*. Aucune compression n'est donc réalisée. Ainsi, il est difficile de savoir à l'avance si ce format donne de bons résultats pour des matrices puisque sa performance dépend entièrement de l'emplacement des éléments non nuls d'une matrice. Par contre, ce format peut être une possibilité pour des matrices dont les éléments non nuls sont rapprochés en groupes permettant ainsi de découper la matrice en blocs majoritairement denses.

2.2.3 Format diagonal

Ce format n'est pas à usage général et ne permet de stocker que les matrices ayant des éléments placés sur un certain nombre de diagonales. La matrice ainsi formatée pourrait stocker toutes ses valeurs non nulles dans un tableau ayant autant de lignes que d'éléments dans la plus grande diagonale et autant de colonnes que de diagonales. De plus, un vecteur regroupe les décalages associés à chaque diagonale par rapport à la diagonale centrale. Par exemple, la matrice *D* suivante est représentée par les tableaux *data* et *offsets* suivants.

$$D = \begin{bmatrix} 11 & 12 & 0 & 0 \\ 0 & 22 & 23 & 0 \\ 31 & 0 & 33 & 34 \\ 0 & 42 & 0 & 44 \end{bmatrix}$$

$$data = \begin{bmatrix} * & 11 & 12 \\ * & 22 & 23 \\ 31 & 33 & 34 \\ 42 & 44 & * \end{bmatrix} \quad offsets = [-2 \quad 0 \quad 1]$$

Figure 2.4 : Exemple de matrice diagonale stockée en format diagonal

Dans leurs expérimentations, Bell et Garland [18] démontrent que ce format offre les meilleures performances en nombre d'opérations à virgule flottante par seconde (FLOPs – *Floating-point Operation per Second*). Par contre, le test est réalisé pour des matrices structurées, où les éléments

non nuls se retrouvent sur une ou plusieurs diagonales. Par contre, le format est inutilisable pour les opérations sur des matrices non structurées.

2.2.4 Format ELLPACK

Le format ELLPACK (ELL) a été créé pour les progiciels ITPACK et ELLPACK permettant de résoudre les systèmes linéaires creux en langage FORTRAN [10]. Il permet de stocker une matrice ayant *M* lignes et *N* colonnes dans deux tableaux de taille *M* par *K*, où *K* représente le nombre maximum d'éléments non nuls que l'on retrouve sur une ligne de la matrice originale. La colonne de chaque élément est donnée par un deuxième tableau de même taille. Par exemple, la matrice *A* suivante est représentée par les tableaux *data* et *indices*.

$$A = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 10 & 0 & 12 & 0 \\ 0 & 21 & 0 & 0 \\ 0 & 0 & 32 & 0 \end{bmatrix}$$

$$data = \begin{bmatrix} 1 & 2 & 3 \\ 10 & 12 & * \\ 21 & * & * \\ 32 & * & * \end{bmatrix} \quad indices = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 2 & * \\ 1 & * & * \\ 2 & * & * \end{bmatrix}$$

Figure 2.5 : Exemple de matrice stockée en format ELLPACK

Tout comme le format diagonal, ce format donne de bonnes performances pour les matrices structurées [18]. Il offre de meilleures performances que le format diagonal lorsque la métrique de comparaison utilisée est la bande passante. Contrairement au format CSR, ce format utilise des matrices denses pour représenter la matrice creuse. Les accès à la mémoire de cette représentation sont plus uniformes pour un processeur parallèle grâce à la possibilité de stocker les éléments en ordre de colonnes (*column-major*) [19]. Plus d'informations sur ce sujet se trouvent à la section 3.3.

Ce format est aussi éliminé lors des tests sur les matrices non structurées. Le problème avec les matrices non structurées est qu'il est possible que certaines lignes de la matrice soient de tailles très élevées et d'autres de tailles très petites. Ainsi, le format ELLPACK peut contenir plusieurs éléments nuls. Un des objectifs principaux de l'utilisation de formats de stockage pour les matrices creuses est l'économie d'espace mémoire. Si la taille des matrices *data* et *indices* sont trop élevées,

le format ELLPACK peut ne pas être avantageux à utiliser en terme de compression de donnée par rapport au stockage de la matrice originale complète.

Représentation diagonale en dent de scie

Un autre format provenant de ITPACK et ELLPACK [10] est appelé diagonal en dent de scie (JAD – *Jagged Diagonal* ou JDS – *Jagged Diagonal Storage*). Ce format requiert une première étape qui ressemble au format ELL puisqu'il faut d'abord décaler les éléments non nuls d'une même ligne vers la gauche. Par contre, les lignes sont ensuite réordonnées en ordre décroissant de tailles. Dans l'exemple de la matrice *A*, les lignes sont déjà dans le bon ordre. Il faut ensuite placer les éléments non nuls dans un vecteur de valeurs en ordre de colonnes (*column-major*), comme à la Figure 2.6.

$$A = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 10 & 0 & 12 & 0 \\ 0 & 21 & 0 & 0 \\ 0 & 0 & 32 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 3 & 0 \\ 10 & 12 & 0 & 0 \\ 21 & 0 & 0 & 0 \\ 32 & 0 & 0 & 0 \end{bmatrix}$$

$$value = [1 \quad 10 \quad 21 \quad 32 \quad 2 \quad 12 \quad 3] \quad column = [1 \quad 0 \quad 1 \quad 2 \quad 2 \quad 2 \quad 3]$$

$$ptr = [0 \quad 4 \quad 6 \quad 7] \quad perm = [0 \quad 1 \quad 2 \quad 3]$$

Figure 2.6 : Exemple de matrice stockée en format JDS

Pour connaître l'emplacement original de chaque valeur non nulle, il faut un vecteur d'indices de colonnes ainsi qu'un vecteur de pointeur de début de colonnes. Enfin, puisque les lignes ont été permutées il faut un vecteur de permutation indiquant les indices de lignes.

Ce format semble compresser la taille originale de la matrice creuse par un plus grand facteur que le format ELL puisqu'aucune valeur nulle n'est stockée. Son utilisation pour la multiplication entre une matrice creuse et un vecteur semble donner un meilleur résultat lors de l'exécution de l'opération. Par contre, la création du format à partir de la matrice originale est plus complexe [20].

2.2.5 Format hybride ELL/COO

Le format hybride reprend le format ELLPACK et améliore sa performance pour des matrices ayant un nombre d'éléments par ligne qui varie énormément. Bell et Garland [18] l'associent donc au format COO pour lequel ce facteur n'a aucun impact.

Le format hybride stocke la majorité des éléments de la matrice sous format ELLPACK et les éléments restants, exceptionnels, dans un format COO. Le nombre de colonnes de la matrice formatée *K* doit être choisi si au moins un tiers des lignes de la matrice originale ont *K* ou plus éléments non nuls. Dans l'exemple de la Figure 2.5, *K* est égal à 3 et un quart des lignes de la matrice *A* ont 3 éléments non nuls. Le nombre qu'il faudrait choisir est 2, car 50% des lignes de la matrice *A* ont 2 ou plus éléments non nuls, ce qui représente plus d'un tiers. Dans ce cas, un seul élément est omis, il est donc stocké dans un format COO.

$$A = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 10 & 0 & 12 & 0 \\ 0 & 21 & 0 & 0 \\ 0 & 0 & 32 & 0 \end{bmatrix}$$

$$data = \begin{bmatrix} 1 & 2 \\ 10 & 12 \\ 21 & * \\ 32 & * \end{bmatrix} \quad indices = \begin{bmatrix} 1 & 2 \\ 0 & 2 \\ 1 & * \\ 2 & * \end{bmatrix}$$

$$row = [0] \;\; column = [3] \; value = [3]$$

Figure 2.7 : Exemple de matrice stockée en format hybride ELL/COO

Si on compare l'exemple de la Figure 2.7 à celui de la Figure 2.5, une économie d'espace est tout de même réalisée. Ce format est avantageux puisqu'il permet d'utiliser le format ELLPACK sans inconvénient ainsi que le plus simple des formats, le format COO. De plus, il est possible d'ajuster sa performance en modifiant la variable *K*. Par contre, le format est plus complexe à réaliser.

2.3 Multiplication d'une matrice creuse et d'un vecteur

Le projet de ce mémoire se concentre surtout sur l'opération de multiplication entre une matrice creuse et un vecteur (SpMV – *Sparse Matrix-Vector Multiplication*). Dans cette section nous présentons l'algorithme ainsi qu'un exemple d'application.

2.3.1 Principes généraux

L'opération de multiplication entre une matrice creuse et un vecteur est à la base de plusieurs équations d'algèbre linéaire. Elle est représentée par l'équation (2.3) où *x* et *y* représentent des vecteurs et *A* une matrice creuse. Pour effectuer cette opération, il faut diviser le problème sur chaque ligne de la matrice creuse, créant des équations indépendantes (2.4). Tout d'abord, il faut multiplier

chaque valeur non nulle d'une même ligne avec un élément du vecteur se trouvant au même indice de colonne. Puis, il faut accumuler ces résultats temporaires dans le vecteur résultat à l'indice de la ligne en question.

$$\vec{y} = A\vec{x} \tag{2.3}$$

$$y[i] = \sum_{j=0}^{M} A[i,j]x[j] \tag{2.4}$$

La multiplication entre une matrice creuse et un vecteur est un problème qui comporte une faible proportion de calculs par rapport au nombre d'accès à la mémoire. Pour chaque opération de multiplication-accumulation, pour des valeurs *i* et *j* fixes représentant un élément non nul, il faut lire l'élément *i,j* de la matrice *A*, l'élément *j* du vecteur *x* et écrire à l'élément *i* du vecteur *y*. Nous avons donc trois accès mémoire pour une opération arithmétique. Nous pouvons penser que la performance de l'algorithme est surtout liée à celle de la mémoire. Pour l'opération de la SpMV complète, le nombre d'accès à la mémoire par rapport au nombre d'opérations peut varier dépendamment de la distribution des éléments non nuls de la matrice *A*, de son format de représentation et du processeur utilisé. Pour un GPU, où chaque instruction effectuée en parallèle sur différentes données ne suit pas un ordre prédéfini, il est possible que chaque lecture et écriture en mémoire se fasse à des adresses complètement différentes et éloignées, créant ainsi de nouveaux transferts de mémoire à chaque opération. Nous verrons ceci plus en détail lorsque nous présenterons le comportement de la mémoire des processeurs graphiques à la section 2.4.2.2.

2.3.2 Algorithme PageRank

Un exemple d'application de la multiplication d'une matrice creuse et d'un vecteur est l'algorithme PageRank. Il permet de déduire l'importance de chaque page web d'un ensemble observé. Page, Brin et al. [3] expliquent l'algorithme avec l'exemple d'un individu qui explorerait aléatoirement un ensemble de N pages web. Sur chacune de ces pages, il a une probabilité *d* de choisir un lien sur la page vers une autre page web et une probabilité (1 – *d*) de quitter la page vers une autre choisie au hasard parmi l'ensemble N (en entrant l'URL directement par exemple). La valeur de *d* est habituellement fixée à 0.85 selon Brin et Page [21].

La variable *d* a été créée pour empêcher les erreurs induites par certains modèles d'interconnexion présents sur le web. L'ensemble des pages sur le web est formé de plusieurs groupes de pages web interreliées qui n'ont aucun lien entre eux. Sans l'ajout de la variable *d* ces groupes accumuleront un haut taux d'importance sans jamais le redistribuer sur le reste de l'ensemble, ce qui empêche les probabilités de l'ensemble total de converger. La variable *d* permet aussi d'éliminer les ambiguïtés créées par des pages ne contenant aucun lien vers d'autres pages web.

La probabilité qu'une page *P* soit visitée par l'individu est la valeur de PageRank associée à cette page web. La probabilité se traduit par la fonction suivante :

$$PR(P) = d\left(\frac{PR(C1)}{L(C1)} + \frac{PR(C2)}{L(C2)} + \cdots + \frac{PR(Ck)}{L(Ck)}\right) + (1-d)\frac{1}{N} \tag{2.5}$$

Où *C1*, *C2*, …, *Cm* représentent les pages citant la page *P* et *L(Ck)* représente le nombre de liens sur la page *Ck*. Ainsi, le vecteur PageRank, représentant toutes les probabilités de l'ensemble de N pages web est représenté comme ceci :

$$\overrightarrow{PR_{\iota+1}} = d\ A\ \overrightarrow{PR_{\iota}} + (1-d)\left[\frac{1}{N}\right]_{N\times 1} \tag{2.6}$$

où la matrice *A* représente les liens de chaque page web vers les autres de la façon suivante :

$$A[i,j] = \begin{cases} {}^{1}\!/_{L(j)} & \text{si j a un lien vers i} \\ 0 & \text{autrement} \end{cases}$$

La matrice A est une matrice creuse puisqu'il est rare qu'une page web ait un lien vers toutes les autres. De plus, tous les éléments d'une même colonne sont égaux et leur somme est égale à 1. Ainsi, les éléments de la matrice sont des nombres rationnels compris entre 0 et 1.

Par exemple, prenons un ensemble de $N = 4$ pages web représenté par la Figure 2.8. Chaque page, représentée par un cercle, détient une série de liens vers les autres, représentées par des flèches. Le nombre associé à chaque lien représente la probabilité de passer de la page source à la page de destination. Cette probabilité est partagée équitablement entre les pages. Ainsi, à partir de la page P3 avec 3 liens, un individu a une chance sur 3 de se retrouver sur la page P1, P2 ou P4. La page P1, quant à elle, ne détenant qu'un seul lien, a une probabilité de 100% de mener à la page P2.

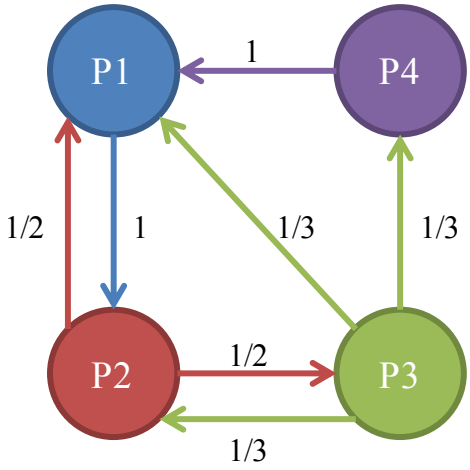


Figure 2.8 : Ensemble de 4 pages web et leurs interconnexions

Les éléments préliminaires au calcul de PageRank, présentés aux équations (2.6) et (2.7), sont les suivants :

$$A = \begin{bmatrix} 0 & 1/2 & 1/3 & 1 \\ 1 & 0 & 1/3 & 0 \\ 0 & 1/2 & 0 & 0 \\ 0 & 0 & 1/3 & 0 \end{bmatrix} \qquad \overrightarrow{PR_0} = \begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix}$$

Nous commençons avec un vecteur d'importance $\overrightarrow{PR}$ égale pour chaque page web. Puis, nous itérons l'équation jusqu'à ce que le vecteur $\overrightarrow{PR}$ converge, c'est-à-dire que $\overrightarrow{PR_i} = \overrightarrow{PR_{i+1}}$. En d'autres mots, $\overrightarrow{PR}$ est le vecteur propre de valeur propre 1 de la matrice A, telle que l'indique l'équation (2.7) définissant le vecteur propre d'une matrice.

$$Ax = \lambda x \tag{2.7}$$

Dans un contexte de PageRank, la recherche du vecteur propre de la matrice est réalisée de façon itérative. On appelle cette méthode la méthode de puissance. Elle permet de ne trouver qu'une seule des solutions possibles au problème, mais est simple à utiliser pour des matrices creuses de très grande taille.

En convergeant, le vecteur $\overrightarrow{PR}$ représente l'importance de chacune des pages web de l'ensemble. Dans notre exemple, la page la moins importante est la page P4 puisqu'elle n'est pointée que par une seule page, P3, et que cette dernière ne lui partage qu'un tiers de son importance. La page la plus importante est la page P2, car elle est pointée par deux pages, P1 et P3, dont P1 lui donnant

toute son importance. Même si la page P1 est pointée par 3 pages, son importance est moindre puisqu'elle ne reçoit qu'une portion de l'importance des pages P2 et P3.

$$\overrightarrow{PR_{27}} = \begin{bmatrix} 0.3328 \\ 0.3763 \\ 0.1974 \\ 0.0934 \end{bmatrix}$$

Selon la précision que nous voulons avoir dans le résultat final, il est possible d'accélérer la convergence en appliquant un critère d'arrêt ε. Afin d'obtenir 27 itérations à notre exemple, la distance Manhattan entre les vecteurs $\overrightarrow{PR}$ des deux dernières itérations doit être plus petite que $5e^{-5}$.

$$\sum_{i=1}^{N} |p_i - q_i| < \varepsilon \tag{2.8}$$

2.4 Processeurs graphiques

Depuis quelques années, les processeurs graphiques sont utilisés pour faire du calcul générique. Pour faciliter la tâche des programmeurs, plusieurs langages de programmation ont été créés pour exécuter ces calculs sur les GPU. Deux d'entre eux, qui ont été étudiés lors de ce projet, sont CUDA (*Compute Unified Device Architecture*) [22] et OpenCL (*Open Computing Language*) [23]. CUDA est une plateforme de programmation parallèle pour les GPU de NVIDIA qui inclut différentes librairies, compilateurs et langages de programmation. CUDA C/C++ permet de programmer les GPU en langage C/C++ augmenté de quelques extensions. OpenCL est un langage ouvert qui permet de programmer en parallèle non seulement tous les modèles de processeurs graphiques, mais aussi les CPU ainsi que tout autre accélérateur offrant une compatibilité OpenCL, comme les FPGA, par exemple. Le choix du langage dépend surtout de la plateforme utilisée et des outils de développement dont le programmeur a besoin. Le langage CUDA est fourni avec plusieurs outils, dont un logiciel d'environnement de programmation, un débogueur et un profileur. Il existe ce genre d'outil pour OpenCL, mais ils sont réalisés par plusieurs groupes dépendamment de la plateforme sur laquelle il sera utilisé. Par exemple, AMD offre un débogueur et un profileur pour les applications en OpenCL [24]. Par contre, puisque CUDA est réalisé spécialement pour les GPU de NVIDIA, la performance offerte sur ces processeurs est plus importante par rapport au langage OpenCL [25].

Le langage CUDA et les processeurs de NVIDIA ont été choisis pour ce projet. Cette section se consacrera à une présentation de ces processeurs nécessaire à la compréhension de ce mémoire. Nous verrons une description du modèle de programmation décrivant la hiérarchie des fils d'exécutions et de la mémoire. Puis, nous décrirons l'architecture générale et les techniques qu'il faut appliquer à une fonction pour qu'elle utilise adéquatement le matériel. Nous finirons, enfin, sur une description de l'architecture Kepler, l'architecture des processeurs qui seront utilisés dans ce projet.

NVIDIA offre trois séries de cartes graphiques : GeForce destinée aux consommateurs, Quadro pour les professionnels et Tesla pour les calculs de haute performance. Cette dernière série est conçue pour être utilisée dans un serveur et supporter une charge de travail soutenue. Les cartes Tesla possèdent des mémoires protégées contre les erreurs (ECC) et deux DMA (*Direct Memory Access*) permettant l'écriture et la lecture simultanées sur le bus PCIe. Les cartes NVIDIA sont aussi classées par versions (*compute capability*) qui vont, présentement, de 1.0 à 5.0. Le nombre majeur de la version correspond aux différentes évolutions de l'architecture des GPU de NVIDIA.

2.4.1 Modèle de programmation

La programmation en CUDA nécessite un système hétérogène utilisant le CPU comme hôte (*host*) et le GPU comme périphérique (*device*). Le code sur CPU s'effectue séquentiellement, alors que le code sur GPU est parallèle. Le système utilisé pour le présent projet est composé d'un CPU, de sa mémoire sur une carte mère, et d'une carte graphique reliée à celle-ci par un bus PCI Express. Une carte graphique est composée du processeur graphique ainsi que d'une mémoire externe.

2.4.1.1 Hiérarchie des fils d'exécution

Les fonctions réalisées sur GPU sont appelées des noyaux (*kernels*) et chacun d'entre eux est exécuté par une multitude de fils d'exécution (*threads*). Les fils sont répartis en groupes appelés blocs (*blocks*) et ceux-ci sont compris dans un plus grand ensemble dénommé grille (*grid*). On lance un noyau en indiquant deux informations : la taille de la grille, représentant le nombre total de blocs, et la taille du bloc, représentant le nombre de fils par bloc. De plus, la taille de la grille et du bloc peut avoir jusqu'à trois dimensions. Dans la Figure 2.9, trois dimensions sont présentées où chaque bloc et fil détient un identifiant. Le nombre de fils et les dimensions permises diffèrent selon les versions. Une description de ces propriétés logicielles se trouve au Tableau 2.2.

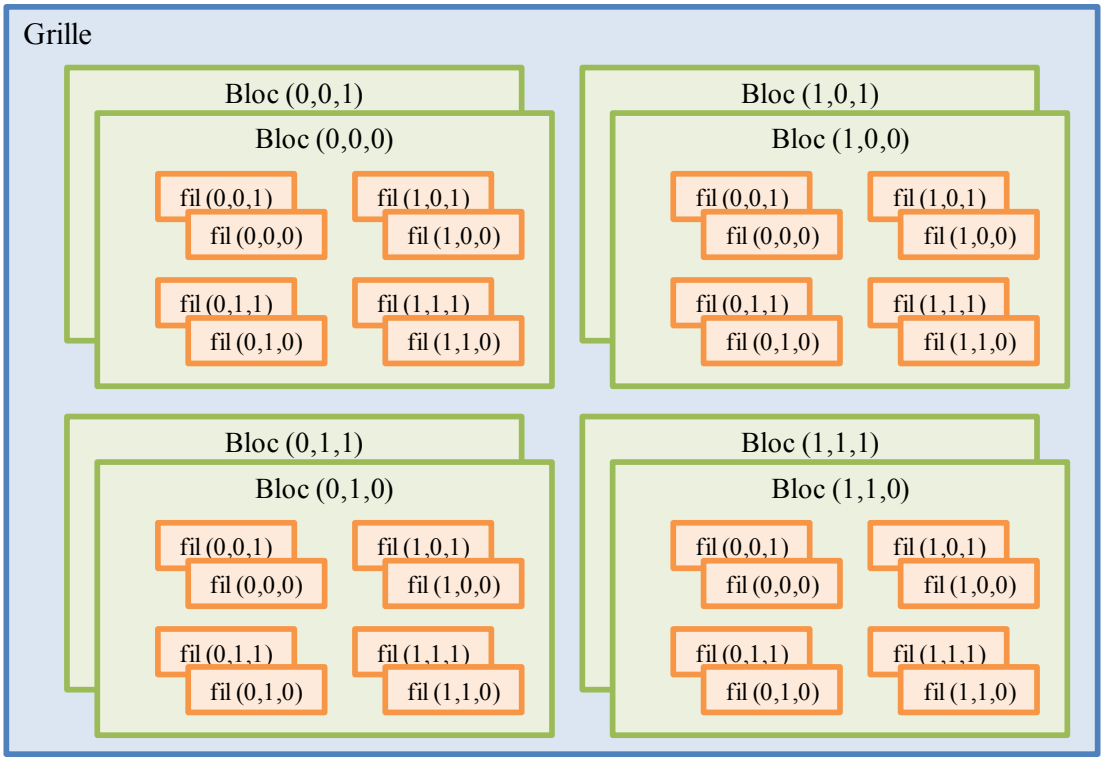


Figure 2.9 : Hiérarchie logicielle des fils d'exécution

Tableau 2.2 : Propriétés maximales logicielles selon les versions d'un GPU NVIDIA

Propriétés	**1.x**	**2.x**	**3.x**	**5.0**
Dimension de la grille	2	3		
Taille de la grille en x	65535		2^{31}-1	
Taille de la grille en y, z	65535			
Dimension d'un bloc	3			
Taille d'un bloc en x et y	512	1024		
Taille d'un bloc en z	64			
Nombre de fils par bloc	512	1024		

Tableau 2.3 : Description des variables intégrées au langage CUDA

Nom	**Type**	**Description**
gridDim	dim3	Dimensions de la grille; nombre de blocs dans chacune des dimensions
blockIdx	uint3	Identifiant du bloc dans chacune des dimensions
blockDim	dim3	Dimensions d'un bloc; nombre de fils dans chaque dimension
threadIdx	uint3	Identifiant du fil à l'intérieur d'un bloc dans chacune des dimensions
warpSize	int	Taille d'une chaîne; nombre de fils par chaîne

Le nombre de fils et de blocs ainsi que leurs identifiants sont accessibles du noyau à l'aide de variables mises à notre disposition par les extensions du langage CUDA. Ces variables sont présentées au Tableau 2.3. Certaines d'entre elles sont de type uint3, représentant un vecteur de trois

nombres entiers positifs et d'autres de type dim3, représentant un vecteur de trois nombres entiers strictement positifs; sans le nombre 0. Un exemple d'utilisation de ces variables est montré à la Figure 2.10, où l'on peut voir une addition entre deux vecteurs. L'identifiant global créé à la ligne 4, à l'aide de variables propres au langage CUDA, permet de référencer chaque fil de façon unique, peu importe le bloc dans lequel il est placé.

```
1   __global__ void monNoyau (int *a, int *b, int *c, int size)
2   {
3     // création d'un identifiant unique pour chaque fil
4     int global_id = blockDim.x * blockIdx.x + threadIdx.x;
5     // on s'assure de ne pas écrire à l'extérieur des limites
6     if (global_id < size)
7       c[global_id] = a[global_id] + b[global_id];
8   }
```

Figure 2.10 : Exemple de noyau en CUDA

2.4.1.2 Hiérarchie de la mémoire

Il existe aussi une hiérarchie dans la mémoire. La mémoire globale est accessible à tous les fils, la mémoire partagée est accessible aux fils faisant partie d'un même bloc et la mémoire locale ainsi que les registres sont accessibles à un seul fil (voir la Figure 2.11).

La mémoire globale est la plus spacieuse, offrant environ quelques gigaoctets pour les versions les plus récentes et n'est pas altérée entre les appels de noyaux d'une même application. Elle se trouve à l'extérieur du GPU et est reliée par PCI express à la carte mère. Par contre, elle détient la plus grande latence du système. Les cartes graphiques de version 3.x, utilisées dans le projet, ont besoin entre 200 et 400 cycles pour effectuer un accès à la mémoire globale.

La mémoire partagée est interne au GPU et se trouve sur chaque multiprocesseur (ces derniers seront discutés en détail à la section 2.4.2). Les versions 3.x de GPU détiennent 48 Ko de mémoire partagée par multiprocesseur. Bien que de taille modeste, cette mémoire se trouve beaucoup plus près des unités de calcul permettant de réduire la latence des demandes en mémoire. En effet, un accès à la mémoire partagée utilise autant de cycles que pour accéder à un registre. Située au même endroit que la cache L1, il est possible de réduire sa taille afin d'avoir une cache L1 plus large.

Enfin, chaque fil a accès à des registres. Toute variable créée dans un noyau est unique à chaque fil d'exécution. Si le nombre de registres par fil est trop grand, si un tableau ou une structure est créé à l'intérieur d'un noyau, les valeurs sont stockées dans la mémoire locale, située au même

niveau que la mémoire globale (à l'extérieur du GPU). Il est donc préférable, dans ces cas, d'utiliser la mémoire partagée.

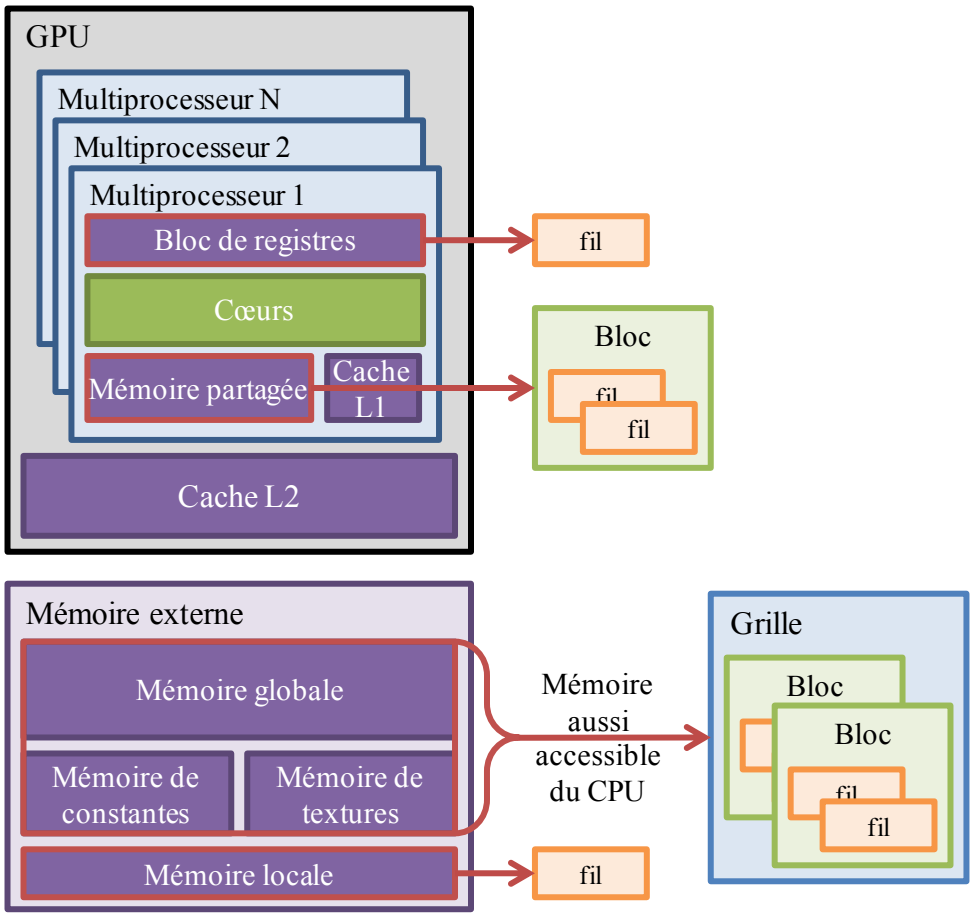


Figure 2.11 : Hiérarchie de la mémoire

Seule la mémoire externe au GPU est accessible avant de lancer un noyau. C'est-à-dire que les paramètres d'un noyau doivent se retrouver sur la mémoire globale, la mémoire de constantes ou la mémoire de textures (utilisée surtout pour l'affichage 3D). Avant l'appel à un noyau, des fonctions CUDA C sont utilisées pour transférer les données à traiter de la mémoire du CPU sur le bus PCIe vers la mémoire du GPU. À la fin de l'exécution du noyau, les données calculées sur GPU doivent être transférées de nouveau vers la mémoire du CPU. On désire alors réduire ce genre de transfert et avoir assez de calculs à réaliser sur le GPU pour que le temps de transfert soit négligeable.

2.4.2 Architecture

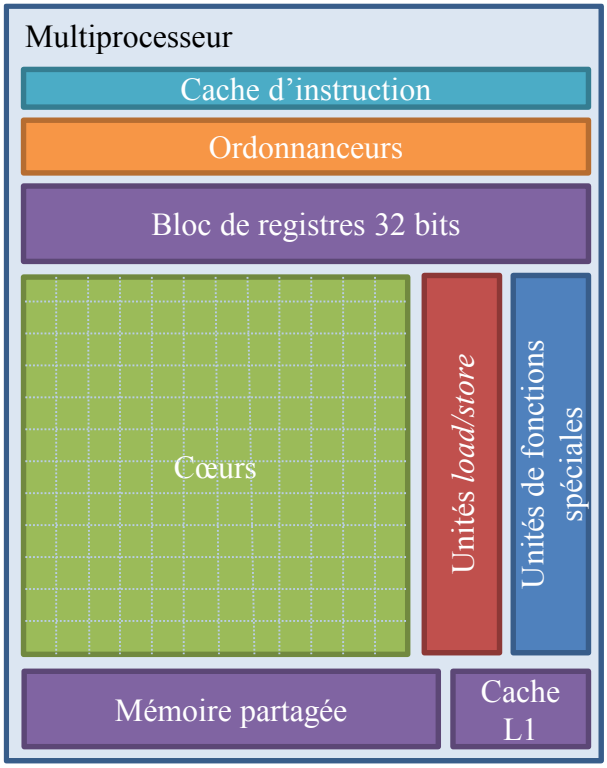


Figure 2.12 : Architecture générale d'un multiprocesseur de GPU

Tableau 2.4 : Caractéristiques par multiprocesseur selon les versions d'un GPU NVIDIA

Propriétés	**1.0**	**1.1**	**1.2**	**1.3**	**2.0**	**2.1**	**3.0**	**3.5**	**5.0**
Nombre de cœurs	8				32	48	192		128
Nombre de registres 32 bits	8 K		16 K		32 K		64 K		
Nombre maximal de registres par fil	128				63			255	
Taille de la mémoire partagée	16 Ko				48 Ko				
Nombre d'unités de fonctions spéciales	2				4	8	32		
Nombre d'ordonnanceurs	1				2		4		

L'architecture interne d'un GPU est composée de plusieurs multiprocesseurs appelés multiprocesseurs de flux (*Streaming Multiprocessors*). Ceux-ci sont chacun dotés d'une série de cœurs CUDA, d'une mémoire partagée et d'un bloc de registre (voir la Figure 2.12). Ils possèdent aussi une cache d'instruction, une série d'ordonnanceurs permettant de distribuer les ressources aux fils d'exécution ainsi que d'autres modèles d'unité de calcul. Les caractéristiques de chacune de ces ressources dépendent de la version du GPU. Le Tableau 2.4 décrit quelques différences d'architecture selon

les versions disponibles. Toutes ces informations se retrouvent en détail dans l'annexe « Compute Capabilities » de la documentation de CUDA [22].

2.4.2.1 Ordonnancement des fils d'exécution

En plus de la hiérarchie logicielle des fils vue à la section 2.4.1, une hiérarchie matérielle est aussi présente. Chaque bloc est d'abord ordonnancé sur les différents multiprocesseurs. Le nombre de blocs pouvant être exécutés en parallèle sur un même multiprocesseur dépend du nombre de registres et de l'espace de mémoire partagée utilisés par les fils dont ils sont composés ainsi que d'une limite imposée par la version du GPU. L'ordre dans lequel les blocs sont exécutés ne peut être prédit à l'avance. Ainsi, il faut s'assurer que chaque bloc soit indépendant des autres, rendant ainsi impossible toute forme de synchronisation entre des fils se trouvant dans deux blocs différents. Par exemple, à la Figure 2.13, un GPU ayant deux multiprocesseurs pourrait ordonnancer le bloc 5 avant le bloc 1. Dans l'exemple, le bloc 1 ne peut s'exécuter que lorsqu'un bloc le précédant, le bloc 5 ou le bloc 2, se termine. Les blocs 5 et 2 sont aussi appelés des blocs actifs au multiprocesseur. C'est-à-dire qu'à n'importe quel moment, des fils d'un de ces deux blocs peuvent être en train d'exécuter une instruction sur ce multiprocesseur.

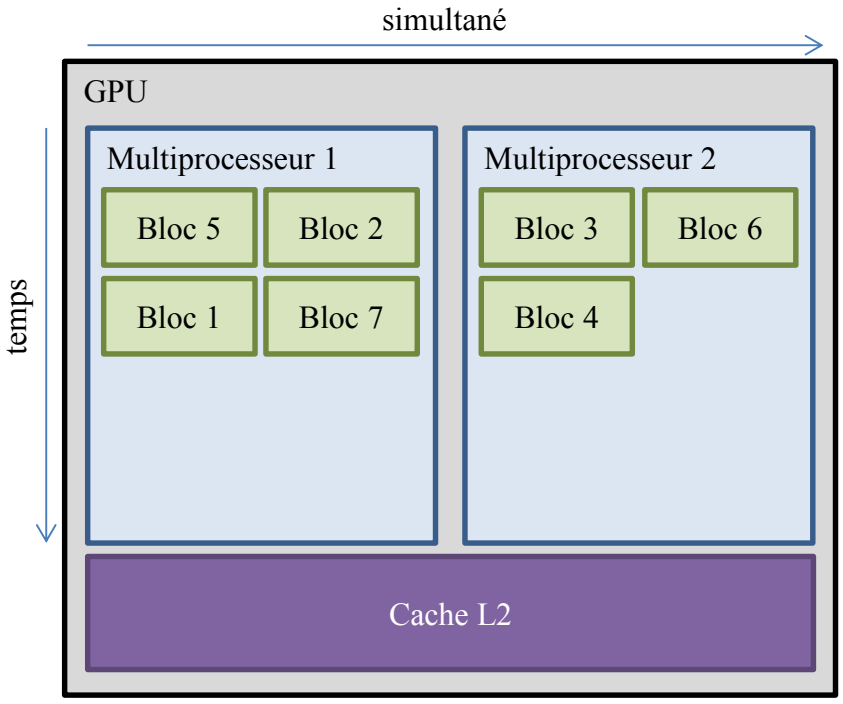


Figure 2.13 : Exemple d'ordonnancement des blocs sur un GPU à deux multiprocesseurs

À l'intérieur d'un multiprocesseur, chaque bloc est séparé en groupes de 32 fils d'exécution appelés chaînes (*warps*). Ces groupes sont toujours exécutés en même temps, c'est-à-dire qu'ils partagent le même compteur de programme. Lorsqu'une chaîne est sélectionnée par un ordonnanceur, 32 cœurs sont alloués, puis tous les fils exécutent la même instruction. S'il y a une divergence entre ceux-ci, causée par exemple par une condition *if/else*, les deux choix seront exécutés l'un à la suite de l'autre. Lorsque la première condition sera exécutée, certains fils seront en attente. Ainsi, certains cœurs seront alloués, mais ne seront pas utilisés. Afin d'avoir une meilleure utilisation des ressources, il est préférable de faire en sorte que les 32 fils d'exécution qui forment une chaîne aient tous la même instruction à exécuter. Parfois, il est plus avantageux de demander à un fil d'exécuter une instruction inutile que de l'empêcher de faire une action. Empêcher un fil d'exécuter la même instruction que les autres fils de la chaîne implique l'inclusion d'une condition de branchement, ce qui peut grandement ralentir le code [22].

De plus, il est à noter que si la taille d'un bloc n'est pas un multiple de 32, la dernière chaîne du bloc sera créée avec le nombre restant de fils. Pourtant, cette chaîne allouera tout de même 32 cœurs lors de son ordonnancement. Il est donc important d'avoir des blocs ayant un multiple de 32 fils afin qu'aucun cœur ne soit inutilisé.

Lorsqu'il y a un appel à la mémoire ou une barrière de synchronisation, l'ordonnanceur choisit une autre chaîne parmi les blocs actifs au multiprocesseur et exécute l'instruction à laquelle elle était rendue (il se peut que ce ne soit pas la même que celle qui a été exécutée au préalable). Le changement d'environnement entre chaînes est réalisé sans temps supplémentaire puisque tous les registres et l'espace de mémoire partagée nécessaire aux blocs actifs sont déjà disponibles dans le multiprocesseur. C'est pourquoi certains blocs sont actifs et d'autre non.

2.4.2.2 Mémoire globale

Les appels à la mémoire globale s'effectuent de façon alignée pour des ensembles de 32, 64 ou 128 octets à la fois, dépendant de la version du GPU. Par exemple, pour des GPU de version 3.x, lorsqu'une chaîne accède à la mémoire, au moins un transfert de 128 octets séquentiel est réalisé dépendant des adresses demandées en mémoire. Si les fils d'une même chaîne demandent des nombres en virgule flottante (4 octets chacun) situés les uns à la suite des autres, une seule transaction est demandée à la mémoire. La Figure 2.14 présente plusieurs cas de requête à la mémoire.

Les rectangles orange (de gauche) représentent les fils et les rectangles mauves (de droite) représentent les éléments en mémoire demandés. Les exemples (a) et (b) requièrent un seul transfert de 128 octets. On appelle ce genre de transfert des accès coalescents à la mémoire. L'exemple (c) de la figure demande des données séquentielles, mais non alignées; c'est-à-dire que le transfert ne s'effectue pas à partir de l'adresse 0. Cet appel à la mémoire nécessitera deux transferts de 128 octets; un appel pour les adresses 0 à 31 et un deuxième pour les adresses 32 à 63, bien que seulement l'adresse 32 soit désirée. Des demandes dispersées à la mémoire de la part d'une même chaîne comme à l'exemple (d) de la figure nécessitent donc plusieurs transferts de mémoire inutiles, ce qui est à éviter dans la mesure du possible.

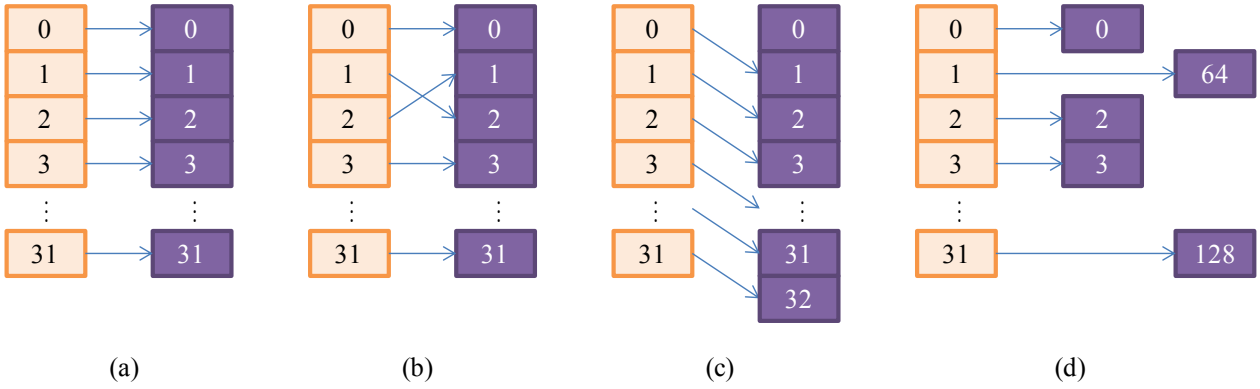


Figure 2.14 : Exemples de demande de transfert d'une chaîne à la mémoire globale

Un accès à la mémoire globale passe par les deux niveaux de cache L1 et L2. Il est possible d'utiliser les deux caches ou seulement la cache L2. Le comportement de chacune des configurations de cache pour les versions 2.0 et plus est comme suit. Dans le cas où l'on utilise les deux caches, la taille d'une transaction est de 128 octets, comme présenté ci-dessus. Dans le cas où l'on utilise seulement la cache L2, les transactions ne sont que de 32 octets. Ceci permet de réduire les transactions inutiles lorsque les adresses à accéder en mémoire sont comme à l'exemple (d).

2.4.2.3 Mémoire partagée

La mémoire partagée est allouée à l'intérieur d'un noyau et peut être visible par tous les fils d'un même bloc. Pour les processeurs de version 2.0 et plus, les données mises en mémoire partagée sont placées dans 32 banques. La largeur des banques est de 32 bits pour les versions 2.x et de 64 bits pour les versions 3.x. Des mots successifs placés dans la mémoire partagée sont assignés dans des banques différentes. Il existe deux modes pour les GPU de version 3.x. Le mode 32-bits assigne

des mots de 32 bits consécutifs à des banques différentes et le mode 64-bits assigne des mots de 64 bits consécutifs à des banques différentes.

Pour utiliser la bande passante maximale lors d'une lecture dans la mémoire partagée, chaque fil d'exécution d'une chaîne ne doit pas demander l'accès à deux éléments différents d'une même banque. Par exemple, pour les GPU de version 3.x en mode 32-bits, un conflit arrive lorsque plus d'un fil demande l'accès à des mots de la même banque qui se trouvent dans deux sections de largeur 64 bits (Figure 2.15). En mode 64-bits, un conflit arrive lorsque plus d'un fil demande l'accès à des mots différents de la même banque (Figure 2.16).

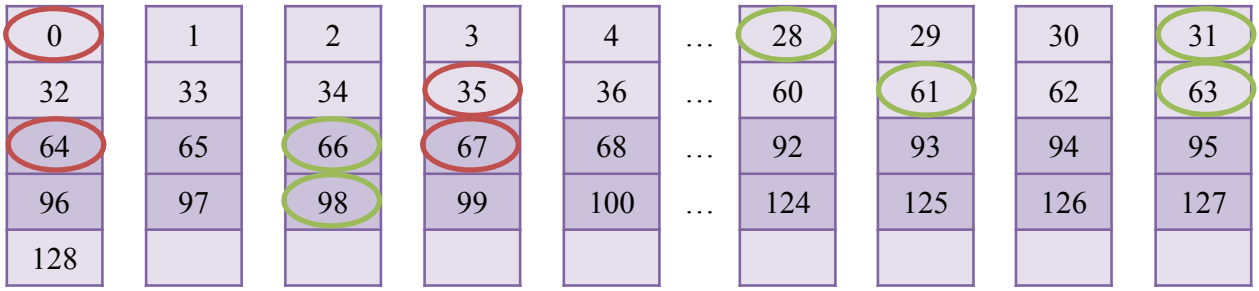


Figure 2.15 : Mots de 32 bits accédés dans la mémoire partagée en mode 32-bits qui occasionnent des conflits (en rouge) ou non (en vert)

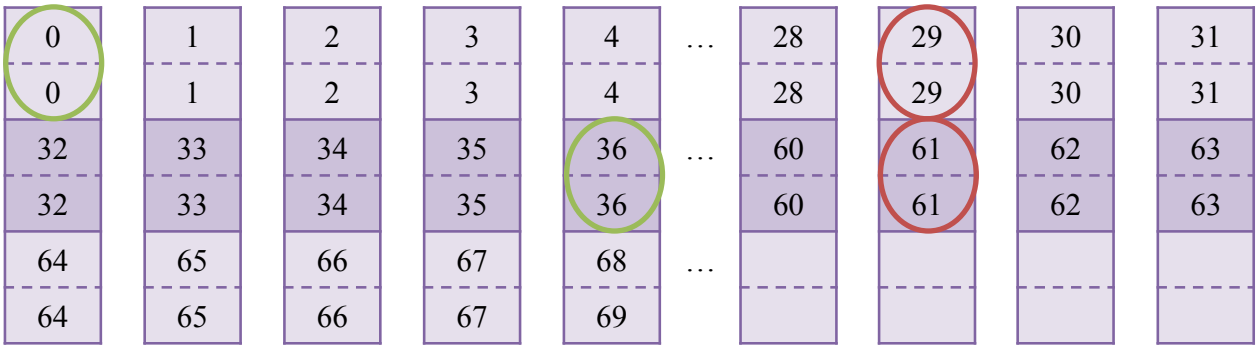


Figure 2.16 : Mots de 64 bits accédés dans la mémoire partagée en mode 64-bits qui occasionnent des conflits (en rouge) ou non (en vert)

2.4.2.4 Architecture Kepler

Ce projet utilise deux cartes graphiques : GeForce GTX 670 de version 3.0 et Tesla K20c de version 3.5. L'architecture des cartes de version 3.x se nomme Kepler. Le changement du nombre majeur de la version occasionne un changement dans l'architecture des multiprocesseurs. Les multiprocesseurs Kepler, appelés SMX, sont dotés de 160 cœurs de plus et de deux fois plus de registres

par rapport à un GPU de version 2.x. Les GPU Kepler ont aussi une fréquence d'horloge plus faible, permettant une consommation énergétique réduite.

Les différences entre les versions 3.0 et 3.5 incluent l'ajout de 64 cœurs dédiés au calcul à virgule flottante double précision, l'introduction d'une technologie de parallélisme dynamique (*Dynamic Parallelism*) et d'une nouvelle architecture de file *Hyper Q* [22].

L'architecture du multiprocesseur de la Tesla K20c est présentée à la Figure 2.17. L'architecture offre quatre ordonnanceurs afin de pouvoir lancer un maximum de quatre chaînes à la fois. Chacun des ordonnanceurs est aussi doté de deux unités de *dispatch*, permettant ainsi, pour la première fois, d'exécuter deux instructions à la fois. Ceci permet donc d'utiliser une certaine parallélisation d'instruction.

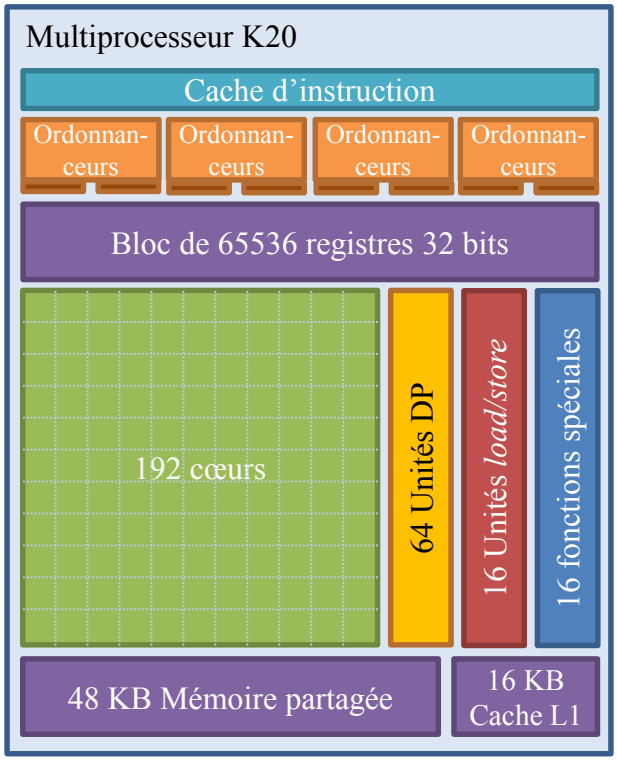


Figure 2.17 : Architecture du multiprocesseur de version 3.5

Le parallélisme dynamique permet aux noyaux d'appeler des noyaux fils ayant leur propre hiérarchie de fils d'exécution. Il permet aussi la récursivité. Ainsi, la communication entre le CPU et le GPU est diminuée et, du même coup, cela réduit le nombre de transferts effectués sur le lien PCIe. Grâce à cette technologie, le GPU devient un coprocesseur plus indépendant, qui prend ses propres décisions.

Les files *Hyper Q* permettent une meilleure distribution des tâches à travers le GPU. Dans les versions précédentes, le GPU ne détenait qu'une seule file de tâches, ordonnançant chacune d'elles une à la suite de l'autre

2.5 Modélisation et implémentation de la SpMV

2.5.1 Approches générales

L'opération de multiplication entre une matrice creuse et un vecteur a été le sujet de plusieurs études tentant de l'accélérer.

Toledo [26] a été l'un des premiers à rassembler tous les problèmes associés à l'exécution d'un tel algorithme. Un facteur principal de la mauvaise performance de la SpMV est le fait que les données sont éparses, ce qui cause beaucoup de manquements de cache. L'auteur a alors proposé de réordonner la matrice afin d'avoir des distances moyennes plus faibles entre deux adresses consécutives. De plus, un système de demande anticipée peut être implémenté pour réduire l'attente des données. Un autre facteur causant la mauvaise performance de l'algorithme est le rapport entre le nombre d'instructions d'accès mémoire par rapport au nombre d'instructions de calcul. Ce problème est traité par l'auteur en séparant la matrice originale en une somme de plusieurs matrices. Des expérimentations sont effectuées avec des matrices placées en format CSR sur des processeurs RISC. Les résultats confirment l'emploi des différentes techniques d'optimisation présentées puisque le nombre d'opérations par seconde en est augmenté.

Pinar et Heath [27] ont aussi remarqué que les accès à la mémoire superflus créés par l'emplacement des éléments de la matrice sont la cause de la pauvre performance d'une SpMV comparée à la multiplication d'une matrice dense avec un vecteur. Ils ont proposé de modifier les structures de stockage des matrices creuses en utilisant un format de compression de lignes par bloc BSR et en réordonnant les éléments non nuls dans un bloc de données. Les résultats sur un serveur Sun Enterprise 3000 montrent une réduction du temps d'exécution de 21 % en moyenne par rapport à l'algorithme conventionnel utilisant le format CSR.

Afin que la SpMV soit optimale sur la machine utilisée, un système, appelé SPARSITY, permettant d'ajuster l'algorithme a été proposé par Im, Yelick et Vuduc [28]. Le système réalise des optimisations au niveau de la réutilisation des données du vecteur x, de la cache et de la structure de stockage des données. Tout comme Pinar et Heath [27], ils ont opté pour un format utilisant des

blocs. En général, SPARSITY permet d'avoir une accélération maximale de 4× par rapport à d'autres microprocesseurs tel que le Pentium III. Leur système aide les utilisateurs à optimiser la SpMV sans qu'ils aient besoin de connaître les détails de la machine utilisée.

2.5.2 Implémentation sur processeurs à multicoeurs

L'opération d'une SpMV est facilement parallélisable puisque les multiplications de chaque ligne de la matrice avec le vecteur sont indépendantes entre elles. Ainsi, plusieurs études se concentrent sur les implémentations parallèles sur des processeurs à multicoeurs.

White et Sadayappan [29] ont observé les problèmes que l'emplacement des données en mémoire cause à l'algorithme de SpMV. Dans leur article, ils mettent l'accent sur l'accessibilité des éléments du vecteur *x* qui dépend entièrement de la structure des éléments non nuls dans la matrice et de la faible proportion entre les instructions de calculs et de lecture en mémoire. Ils ont donc proposé de paralléliser l'algorithme en augmentant le nombre d'instructions de calcul par déroulement de boucle tout en évitant d'empirer la localité des éléments en mémoire. Les résultats ont démontré que le format BSR performe mieux en termes du nombre d'opérations à la seconde comparé au format CSC et à une version compressée par bloc de ce dernier. Le format de compression par bloc du format CSC a toutefois surpassé les performances du format BSR lorsque les matrices avaient plusieurs lignes ayant peu d'éléments non nuls.

Une comparaison de plusieurs processeurs à multicoeurs exécutant une SpMV est présentée par Williams et al. [30]. Ils ont présenté des optimisations possibles telles que la division équitable des tâches pour chaque fils, l'utilisation de la cache pour certains vecteurs, dont *x* ou *y* et l'utilisation des formats BCOO (*block coordinate*), ou BSR, au lieu du format CSR, afin de diminuer la taille des matrices. Dans leur système, des optimisations sont choisies au préalable, puis la matrice est formatée et le noyau correspondant est généré. Sur les processeurs ayant de deux à quatre cœurs plus complexes, l'implémentation optimisée surpasse le nombre d'opérations à virgule flottante d'une implémentation naïve utilisant le format CSR d'un facteur d'environ 1.5×. La performance de ces processeurs dépend grandement de la structure de la matrice testée. Les processeurs ayant de 8 à 16 petits cœurs ont performé jusqu'à 6× et 13× mieux en utilisant le plus de cœurs possible. Les auteurs en ont donc conclu que, pour des algorithmes tels que la SpMV, le nombre de cœurs est plus important que leurs performances individuelles et que les processeurs doivent se concentrer à maximiser leur bande passante à la mémoire. Ainsi, le GPU serait une bonne alternative aux CPU.

Une implémentation de la SpMV a aussi été présentée sur les processeurs de 60 cœurs Intel Xeon Phi par Liu et al. [31]. Les auteurs ont identifié plusieurs problèmes à l'algorithme utilisant le format CSR, dont la difficulté d'utiliser une architecture SIMD adéquatement à cause des différentes tailles de lignes de la matrice et les accès à la mémoire irréguliers. Ils ont donc proposé un nouveau format de compression par blocs basé sur le format ELLPACK. Ce format leur a permis d'avoir plus d'opérations à la seconde d'un facteur d'en moyenne 3.52× et 1.32× que la meilleure implémentation sur Intel Xeon E5-2680 et NVIDIA Tesla K20X, respectivement.

2.5.3 Implémentation sur FPGA

Un des premiers travaux à essayer la mise en œuvre d'une SpMV sur FPGA a été présenté par DeLorimier et DeHon [32]. Ils ont remarqué que l'exécution de cet algorithme ne peut pas atteindre la meilleure performance de calcul sur les processeurs à multicoeurs, la principale limitation étant la bande passante. Pour remédier au problème, ils ont proposé de porter l'algorithme sur plusieurs FPGA, des plateformes qui ont une bande passante plus élevée que les multicoeurs, en général. Leur implémentation utilise le format CSR et divise l'opération en une série de produits scalaires entre une ligne de la matrice et le vecteur x. Chaque élément de traitement est muni d'un multiplieur-accumulateur et permet de calculer un élément du vecteur de destination y. L'architecture a été testée sur différentes matrices et a été comparée à d'autres processeurs à multicoeurs. Le nombre d'opérations à la seconde de leur implémentation dépasse toujours celui des CPU testés. De plus, la fraction de la performance obtenue par rapport à la meilleure performance attendue du processeur est toujours plus élevée pour les implémentations sur FPGA. Ceci signifie que moins de performance est perdue en utilisant une opération telle que la SpMV lorsqu'elle est exécutée sur un FPGA.

Un autre design sur FPGA est présenté par Zhuo et Prasanna [33] utilisant un format CSR paramétrable. Ils ont été motivés par les faibles améliorations qui ont été réalisées en optimisant l'algorithme [27], [28]. Leur architecture consiste en un arbre binaire dont le nombre de feuilles est paramétrable. Les feuilles s'occupent de la multiplication entre un élément non nul de la matrice et un élément du vecteur. Chaque étage suivant s'occupe d'additionner deux résultats temporaires adjacents. Le nombre de feuilles est paramétrable et la performance augmente lorsque ce nombre augmente. Leurs résultats ont démontré que le nombre d'opérations par seconde de cette architec-

ture est 1.5× plus élevé que la valeur de la bande passante du FPGA cible. Comparée à une implémentation sur CPU utilisant SPARSITY [28], leur implémentation sur FPGA performe mieux, surtout pour des matrices ayant une structure très irrégulière.

Dans un contexte d'application, McGettrick, Geraghty et McElroy [34] ont présenté une architecture sur FPGA pour l'algorithme PageRank. Leur architecture est de type système sur puce et utilise un MicroBlaze pour lui permettre d'être utilisé pour n'importe qu'elle application utilisant une multiplication entre une matrice creuse et un vecteur. Les auteurs ont comparé leurs résultats à la performance de l'algorithme sur un CPU. Les résultats sont concluants, atteignant parfois une performance deux fois plus grande même si l'horloge du FPGA est 10 fois plus lente.

Enfin, Zhang et al. [35] ont décidé de comparer les FPGA et les GPU par rapport à la SpMV. Compte tenu de plusieurs travaux antécédents vantant les mérites des GPU comparés aux FPGA, ils ont proposé une nouvelle implémentation de la SpMV sur FPGA. Leur architecture sépare l'opération en cinq blocs indépendants s'occupant de multiplier une ligne de la matrice avec le vecteur x. Les résultats ont montré que le GPU testé surpasse toujours leur implémentation sur FPGA. En effet, le GPU testé a une bande passante plus élevée que leur FPGA. Par contre, les auteurs ont estimé qu'à bande passante égale, le FPGA performerait mieux. Ils ont aussi remarqué que la bande passante du GPU varie avec la taille de la matrice en mémoire. Ceci est dû au fait que plus il y a de fils d'exécution en parallèle, plus il est possible pour le GPU de masquer les temps d'accès à la mémoire avec des instructions de calcul, augmentant ainsi la bande passante atteignable. Par contre, cela fait en sorte que la performance observée avec les petites matrices est moins bonne. Le FPGA, quant à lui, a cet avantage de toujours pouvoir utiliser la bande passante maximale pour tout accès à la mémoire.

2.5.4 Implémentation sur GPU

L'utilisation des GPU pour réaliser des calculs d'ordre général est devenue plus populaire au fil des années. Plusieurs travaux récents portent alors sur l'accélération de la SpMV sur GPU.

Bell et Garland [18] ont présenté différentes représentations de matrices creuses ainsi que leurs implémentations respectives de la SpMV en langage CUDA. Les formats de représentation présentés sont ceux qui se retrouvent à la section 2.2, ainsi qu'un format appelé Paquet (PKT). Deux ensembles de matrices ont été utilisés, des matrices dites structurées qui ont une forme diagonale

et des matrices dites non structurées dont l'emplacement des éléments non nuls ne répond pas à une structure de matrice connue. Pour le premier ensemble, le format DIA performe le mieux en termes du nombre d'opérations à virgule flottante par seconde. Pour le deuxième ensemble, c'est le format HYB qui performe le mieux.

Pour le format CSR en particulier, Baskaran et Bordawekar [36] ont présenté plusieurs optimisations. Ils ont pris en compte différents aspects : les problèmes de synchronisation entre les fils d'exécution, la distribution du travail entre les fils, les accès à la mémoire globale et la réutilisation des données. Ils ont comparé leurs résultats à ceux obtenus avec le format HYB [18] et, dépendamment du nombre d'éléments non nuls par ligne de la matrice, leur implémentation performe jusqu'à 15% mieux. Ainsi, non seulement il faut choisir le bon format, il faut aussi optimiser l'implémentation du noyau.

Un autre exemple d'optimisation de la distribution du travail entre les fils d'exécution est présenté par Wu et al. [37] sur des processeurs graphiques AMD. Ils ont testé plusieurs configurations de distribution des tâches et c'est celle qui utilisait un mélange entre un et plusieurs fils par ligne de la matrice qui a donné le plus grand nombre d'opérations à virgule flottante par seconde.

Plusieurs auteurs proposent plutôt de créer un tout autre format de représentation pour accélérer l'algorithme de multiplication entre une matrice creuse et un vecteur. C'est le cas de Vázquez et al. [19] qui ont proposé le format ELLPACK-R. Ce format est identique au format ELL, à la différence qu'un vecteur est ajouté pour stocker les tailles de chaque ligne de la matrice. Ceci permet alors de tirer avantage du format ELL, qui offre de meilleures performances que le format usuel CSR dans plusieurs cas, sans perdre de temps sur les éléments nuls qui se trouvent dans les matrices *data* et *indices* de la représentation du format ELL. Vázquez et al. ont aussi proposé une modification de l'implémentation de ce format sur une plateforme parallèle nommée ELLR-T [38] où T représente le nombre de fils d'exécution qui s'occupent d'une ligne de la matrice creuse. Monakov et al. [39] ont présenté un format nommé ELLPACK en tranches (*sliced ELLPACK*) qui sépare le format ELL en sections formées de lignes consécutives ayant chacune un K différent afin de tenter de régler le même problème que le format précédent en stockant moins d'éléments nuls dans la mémoire. Puis, à partir de tous ces formats, Dziekonski et al. [40] ont proposé un format nommé ELLR-T en tranches (*sliced ELLR-T*). Le format permet alors de stocker moins d'éléments nuls

comme le format ELLPACK en tranche, permettant ainsi d'avoir besoin de presque aussi peu d'espace de mémoire que le format CSR selon les matrices testées. De plus, le format permet d'accélérer l'exécution de l'algorithme en omettant de réaliser les opérations de multiplication sur les éléments nuls, tout comme le format ELLPACK-R, et en assignant plusieurs fils d'exécution à une même ligne, tout comme le format ELLR-T.

Un autre format ayant fait l'objet d'améliorations est le format COO. Dang et Schmidt [41] ont proposé le format COO en tranches (*sliced COO*). Ce format est obtenu en tranchant le format COO en sections de lignes consécutives. Les lignes sont placées en ordre de longueur au préalable afin que des fils d'exécution consécutifs aient une charge de travail semblable. Les éléments non nuls de la même colonne sont placés les uns à la suite des autres dans le vecteur de valeurs afin d'avoir des appels à la mémoire plus coalescents. Le format a été comparé aux formats de la librairie CUSP [14] et il donne de bons résultats en général. Pour les opérations à simple précision, le format COO en tranches est exécuté en moins de temps que le format COO, CSR et HYB sur des matrices non structurées [18].

Une autre façon d'accélérer la SpMV est de réduire la taille des données à accéder en mémoire. Tang et al. [42] ont proposé pour cela de compresser les formats de représentation avant de les placer dans la mémoire du GPU. Afin de réaliser une multiplication entre un élément non nul de la matrice et le vecteur x, le format COO doit lire quatre adresses en mémoire et le format ELL trois. Réduire la taille des données à lire permettrait de rééquilibrer le rapport élevé du nombre d'instructions de mémoire par instruction de calcul. Ainsi, les auteurs ont compressé les formats ELL, COO et HYB et, par rapport aux formats non compressés, des économies d'espaces allant parfois jusqu'à 90 % et une accélération en moyenne de 1.5× ont été observées.

Choi et al. [43] ont présenté un article qui propose un nouveau format appelé BELLPACK. Ce format est similaire au format ELL, mais il regroupe les éléments non nuls en blocs comme le format BSR. De plus, ces auteurs ont proposé un modèle permettant d'ajuster les paramètres de leur format afin que leur implémentation finale soit optimale et qu'elle offre le temps d'exécution le plus court. Leur modèle estime le temps d'exécution avec une précision d'en moyenne 86.5 %. Les tests qui ont permis d'estimer cette précision ont été réalisés sur des matrices qui formaient des blocs de valeurs non nulles les plus denses possible, c'est-à-dire qu'ils ont environ le même nombre d'éléments non nuls sur chaque ligne.

2.5.5 Implémentation sur GPU avec modélisation

Les tentatives d'accélération de la SpMV et les performances différentes observées avec différentes matrices creuses, différents formats ou implémentations ont amené certains auteurs à considérer la modélisation du problème.

Avant de se pencher sur un modèle de la SpMV, il faut un modèle du processeur graphique sur lequel l'algorithme sera exécuté. Hong et Kim [44] ont étudié le fonctionnement des GPU en général afin de créer un modèle permettant de prédire le temps d'exécution de n'importe quel algorithme. Dans ce modèle, deux variables ont été prises en compte : le parallélisme de chaînes par rapport à la mémoire et par rapport aux calculs. La première métrique représente le nombre de chaînes par multiprocesseur qui peuvent accéder à la mémoire en même temps et la deuxième métrique représente le nombre de chaînes par multiprocesseur qui peuvent réaliser un calcul pendant une phase d'accès à la mémoire. Les résultats de leurs expérimentations ont présenté une erreur maximale de 13.3 % entre les temps d'exécution estimé et mesuré. Les algorithmes utilisés par les auteurs pour valider leur modèle sont simples et les caractéristiques nécessaires, comme le nombre d'accès à la mémoire, sont faciles à trouver. Pour ce qui est de la SpMV, le nombre d'accès à la mémoire est difficile à prédire. Il faut donc une étape de plus au modèle de Hong et Kim afin qu'il puisse être utile pour prédire la performance d'une SpMV.

Par rapport aux accès mémoire, He et al. [45] ont présenté un modèle de performance des opérations de lecture et d'écriture pour un GPU. Ils ont remarqué que lorsque les accès à la mémoire sont irréguliers, la bande passante utilisée est diminuée et la latence est augmentée. Ils ont donc modélisé la bande passante effective pour des accès mémoire séquentiels et aléatoires permettant d'estimer le temps de transfert d'un ensemble de données. En moyenne, le temps estimé par leur modèle atteint une précision de 87 % par rapport au temps mesuré d'une lecture en mémoire et une précision de 90 % pour une écriture en mémoire.

Utilisant une approche expérimentale, El Zein et Rendell [46] ont essayé d'identifier la meilleure implémentation CUDA pour réaliser une SpMV en utilisant le format CSR. Pour ce faire, ils ont comparé plusieurs noyaux générés automatiquement qui utilisent différentes options. Par exemple, une combinaison serait d'utiliser une distribution du travail afin que chaque ligne de la matrice soit prise en compte par une chaîne de fils d'exécution dans une implémentation où les éléments des trois vecteurs de la représentation en format CSR sont placés dans la mémoire de texture et qu'ils

utilisent des types float4 représentant quatre nombres à virgule flottante. Un test exhaustif de toutes ces combinaisons génère un total de 324 différents noyaux qui ont été exécutés sur un ensemble de 735 matrices creuses. Les résultats font ressortir les quatre noyaux les plus performants. Un dernier modèle basé sur un arbre décisionnel a alors été proposé pour choisir la meilleure implémentation parmi les quatre meilleures pour une matrice cible.

Récemment, Guo, Wang et Chen [47] ont présenté un modèle de performance à la fois analytique et expérimental ainsi qu'un outil d'optimisation pour réaliser une SpMV sur GPU. Le modèle exécute une SpMV sur une série de matrices générées automatiquement dans un format et un GPU cible. Les temps d'exécution recueillis par profilage sur ces matrices ont permis de créer des relations entre le temps d'exécution et deux caractéristiques : le nombre d'éléments non nul par ligne et le nombre de fils pouvant rouler en parallèle sur le GPU cible. Ces relations peuvent aussi prédire le temps d'exécution pour n'importe quelle matrice. Les résultats de leurs travaux montrent que la différence entre le temps d'exécution estimé et mesuré est de 9 % ou moins.

Dans ce projet, nous proposons un modèle permettant de mesurer le nombre de transactions à la mémoire. Nous démontrerons qu'il est possible d'utiliser ce nombre afin d'estimer la performance d'une implémentation d'une SpMV par rapport à une autre. Le modèle est totalement analytique contrairement à ceux proposés par El Zein et Rendell [46] et Guo et al. [47].

2.5.6 Bilan

Les GPU semblent être des plateformes qui surpassent les CPU puisqu'ils sont dotés de plus de coeurs, d'une architecture parallèle et d'une bande passante plus élevée. Par contre, certains auteurs observent les limitations des GPU par rapport aux CPU.

C'est le cas de Vuduc et al. [48] qui ont analysé les limites des GPU pour trois domaines, dont la résolution des systèmes linéaires creux. Sur ce sujet, les auteurs ont démontré que bien que leurs travaux sur l'accélération de la SpMV sur GPU leur a permis d'atteindre une performance deux fois plus élevée que sur les CPU, plusieurs aspects ne sont pas pris en compte. Par exemple, le transfert des données du CPU vers le GPU est une étape importante du processus de la SpMV qui est souvent omis dans les résultats. Pourtant, ce transfert est l'un des plus lents du système hétérogène CPU/GPU. De plus, plusieurs optimisations sur GPU nécessitent une réorganisation des don-

nées avant le transfert. Le temps d'exécution de cette étape est aussi omis lors du calcul des résultats. Leur article conclut sur le fait que pour faire une vraie différence en utilisant les GPU, il faut que les comparaisons soient réalistes.

Également, Davis et Chung [49] ont décidé de se concentrer sur les avantages d'utilisation des processeurs graphiques par rapport aux CPU pour accélérer une SpMV. Ils analysent la performance, la puissance consommée et le coût de chaque implémentation. Les GPU sont présentés comme des processeurs très performants ayant plusieurs cœurs, une architecture parallèle et une grande bande passante à la mémoire. Par contre, il est difficile d'atteindre de tels gains de performance sans d'abord optimiser l'algorithme de SpMV. De plus, de plus en plus de nouveaux CPU se rapprochent des capacités des GPU et ont définitivement plus d'espace mémoire. Les auteurs ont réalisé des expérimentations sur deux Xeon et deux GPU de NVIDIA. Le code utilisé provient de Williams et al. [30] pour le CPU et la librairie CUSP [14] pour le GPU. Les résultats montrent que les GPU sont entre 2 et 3 fois plus rapides que les CPU. Par contre, plusieurs matrices sont trop grandes pour être placées dans la mémoire des GPU et la taille des matrices utilisées dans des problèmes réels risque d'augmenter de plus en plus.

Finalement, les auteurs présentés dans cette revue de littérature s'entendent en général pour dire que l'exécution d'une SpMV est ralentie par les accès à la mémoire très dispersés qui sont causés par l'emplacement des éléments non nuls de la matrice creuse et qu'il est difficile de prévoir le temps d'exécution final. Le GPU semble être la plateforme de choix pour une telle opération bien qu'elle comporte quelques limitations.

CHAPITRE 3 MODÈLE PROPOSÉ

Ce chapitre présente le modèle proposé dans le cadre de ce projet de recherche. La première section présente la méthodologie et les variables utilisées dans le modèle. Les quatre sections suivantes présenteront les équations et algorithmes utilisés pour mesurer le nombre de transactions et le nombre de requêtes à la mémoire. Cinq différentes implémentations seront présentées : deux implémentations utilisant le format CSR, puis une implémentation pour chacun des formats ELL, COO et HYB.

3.1 Méthodologie

Le modèle proposé dans ce chapitre permet de calculer le nombre d'accès à la mémoire en effectuant une analyse du code ainsi que de la matrice cible. Le modèle utilise une méthode semblable à un profilage statique puisqu'il analyse les instructions utilisées dans le code de l'implémentation utilisée sans l'exécuter. Le modèle utilise aussi des informations de la matrice creuse cible, dont sa taille et l'emplacement des éléments non nuls qui la compose.

Ce chapitre utilisera les variables présentées au Tableau 2.1, présentant les paramètres d'une matrice creuse, et au Tableau 3.1.

Tableau 3.1 : Variables utilisées dans le modèle proposé

Variables	**Description**
a	Nombre d'éléments manquants à l'alignement
I_R	Nombre d'instructions demandant une requête d'accès à la mémoire
N_e	Nombre d'éléments demandés en mémoire
N_t	Nombre de fils
N_w	Nombre de chaînes
R	Nombre de requêtes à la mémoire
rem	Nombre de fils dans la dernière chaîne
S_b	Taille d'un bloc; nombre de fils dans un bloc
S_e	Taille, en octet, des éléments en mémoire
S_{tx}	Taille d'un transfert en octets; taille d'une ligne de cache
S_w	Taille d'une chaîne; nombre de fils dans une chaîne
T	Nombre de transactions à la mémoire
W	Nombre de chaînes

La variable *Srow*, du Tableau 2.1, peut être facilement calculée au préalable pour chaque matrice. Puisqu'elles sont d'abord toute en format COO, il est possible d'utiliser le vecteur *row* afin de construire le tableau *Srow* comme à la Figure 3.1. Dans le vecteur *row*, le nombre d'occurrences

de chaque indice de ligne est additionné et le total est répertorié dans le vecteur *Srow* à l'indice correspondant. La variable *maxRow*, qui représente la plus longue ligne, correspond alors à l'élément le plus élevé du tableau *Srow*.

```
1    for i = 0 to N_NZ-1 do
2       Srow[row[i]] ++
3    end for
```

Figure 3.1 : Calcul du nombre d'éléments de chaque ligne de la matrice

Les accès à la mémoire peuvent être mesurés en nombre de transactions et en nombre de requêtes. Une requête représente une demande de lecture ou d'écriture par les fils faisant partie d'une même chaîne. Comme présenté à la section 2.4.2.2 portant sur la mémoire globale, une requête demandant des adresses séquentielles correspondant à une région alignée de la mémoire est plus efficace. C'est-à-dire qu'elle requiert moins de transactions à la mémoire.

Pour mesurer le nombre de requêtes, il est nécessaire de connaître le nombre d'instructions effectuant une demande d'accès à la mémoire ainsi que le nombre de chaînes de fils d'exécution effectuant cette instruction (3.1).

$$R = I_R W \tag{3.1}$$

Une requête a besoin d'une seule transaction à la mémoire lorsque la taille de la région demandée est inférieure ou égale à la taille d'un transfert et que la première adresse est alignée (3.2). Une requête est alignée lorsque la première adresse de l'ensemble demandé est un multiple du rapport de la taille de transfert S_{tx} par la taille des éléments S_e. En principe, ce rapport est un entier.

$$T = \left\lceil \frac{(N_e + a) \times S_e}{S_{tx}} \right\rceil \tag{3.2}$$

Le calcul du nombre de transactions ne tiendra pas compte de la cache dans ce chapitre. Par exemple, si deux instructions qui se suivent dans un noyau requièrent des adresses similaires, le nombre de transactions total de chaque instruction sera calculé indépendamment. En effet, bien que deux instructions se suivent dans le noyau, cela ne veut pas dire qu'elles seront exécutées l'une à la suite de l'autre. Il n'est pas possible de prédire l'ordre dans lequel les chaînes de fils d'exécutions seront lancées. Il se peut que suite à une instruction, une autre chaîne qui a besoin d'effectuer

une instruction totalement différente soit lancée. Nous estimons donc qu'à chaque fois, les éléments mis dans la cache se font écraser par d'autres.

Puisque la SpMV requiert beaucoup d'accès à la mémoire, nous posons l'hypothèse que le temps d'exécution est lié au nombre de transferts à la mémoire *T*. Nous proposons d'estimer la performance de la SpMV selon différents formats de représentation à l'aide du nombre de transactions à la mémoire. Une implémentation ayant un plus grand nombre de transactions qu'une autre est moins performante que cette dernière.

Dans les sections qui suivent, nous présenterons les équations permettant de calculer le nombre de transactions à la mémoire de 5 implémentations de la SpMV. Deux implémentations seront présentées pour le format CSR, la première ayant une distribution des tâches où chaque fil est associé à une ligne de la matrice et une deuxième où une chaîne de fils est associée à une ligne. Ensuite, nous verrons une implémentation pour chacun des formats ELL, COO et HYB. Ces formats ont été choisis de façon à ce qu'ils soient compatibles avec n'importe quel type de matrice. De plus, ce sont des formats utilisés dans les bibliothèques de résolution de problèmes avec matrices creuses basées sur le langage CUDA : CUSP [14] et CUSPARSE [13]. Les implémentations présentées sont d'ailleurs inspirées de ces bibliothèques.

Pour chacune des implémentations, le nombre de transactions et de requêtes à la mémoire sera scindé en une somme de plusieurs nombres comme le montre les équations (3.3) et (3.4). La valeur maximale de i varie d'un format à l'autre. Les nombres de transactions de T_i et R_i sont chacun associés à un différent vecteur de données à accéder en mémoire. Dans certains cas, ces nombres de transactions et de requêtes peuvent être calculés par une expression analytique et dans d'autres cas, ils sont calculés par profilage statique, c'est-à-dire que leur valeur est calculée par un code essayant de reproduire le comportement de l'implémentation de la SpMV.

$$T_{format} = \sum_i T_i \tag{3.3}$$

$$R_{format} = \sum_i R_i \tag{3.4}$$

3.2 Format CSR

Dans cette section, nous calculerons le nombre de transactions et requêtes à la mémoire de deux implémentations de la SpMV utilisant des matrices en format CSR.

Le format CSR décrit une matrice creuse à l'aide de trois vecteurs. Deux vecteurs représentent les indices des valeurs et des colonnes de la matrice. Le troisième vecteur représente l'emplacement, dans le vecteur des valeurs, des éléments non nuls en tête de ligne. Deux éléments adjacents du vecteur de pointeur de ligne représentent donc les indices du début et de fin de ligne (voir section 2.2.2). Les nombres de transactions et de requêtes de cette section seront scindés en plusieurs sections représentant chaque vecteur à accéder en mémoire. Ainsi, le nombre de transactions T sera représenté par le nombre de transactions aux vecteurs de valeurs, T_v, de colonnes, T_c et de pointeur de ligne, T_{ptr}, de la matrice en format CSR ainsi que du vecteur x, T_x, avec lequel la matrice est multipliée et le vecteur y, T_y, où sont stockés les résultats (3.5). Le nombre de requêtes sera divisé de la même façon.

$$T_{CSR} = T_{ptr} + T_v + T_c + T_x + T_y \tag{3.5}$$

Dépendamment de la façon dont les tâches sont distribuées aux multiples fils d'exécution, la SpMV peut être implémentée avec un fil d'exécution associé à chaque ligne ou une chaîne de fils associée à chaque ligne.

3.2.1 Un fil d'exécution par ligne

Le noyau effectuant une SpMV utilisant un fil par ligne d'une matrice en format CSR est présenté à la Figure 3.2. Ce code est basé sur le noyau nommé « *scalar kernel* » présenté par Bell et Garland [18].

Chaque élément du vecteur de résultats de la multiplication entre une matrice et un vecteur peut être calculé indépendamment, comme présenté à l'équation (2.4). La façon la plus simple de paralléliser l'opération est d'assigner un fil d'exécution pour le calcul de cette équation.

Comme présenté à la section 2.4.1, un noyau est exécuté par une multitude de fils sur le GPU. Les fils sont exécutés par groupes appelés chaînes ayant un même compteur de programme. Une instruction du pseudocode à la Figure 3.2 est exécutée par une chaîne, soit un groupe de S_w fils. À la ligne 4, le nombre de fils est restreint à ceux dont l'identifiant correspond à une ligne de la matrice.

Ainsi, comme indiqué par l'équation (3.6), le nombre de chaînes exécutant les instructions qui suivent est égal au nombre M d'éléments dans le vecteur de résultats, y, divisé par le nombre de fils par chaîne S_w.

$$W = \left\lceil \frac{M}{S_w} \right\rceil \tag{3.6}$$

```
 1   int ligne = blockIdx.x * blockDim.x + threadIdx.x;
 2   float somme = 0.0;
 3
 4   if (ligne < M)
 5   {
 6       debut = ptr[ligne];
 7       fin = ptr[ligne + 1];
 8
 9       for (int i = debut; i < fin; i++)
10           somme += val[i] * x[col[i]];
11
12       y[ligne] = somme;
13   }
```

Figure 3.2 : Code du noyau effectuant une SpMV utilisant le format CSR et attribuant un fil par ligne de matrice

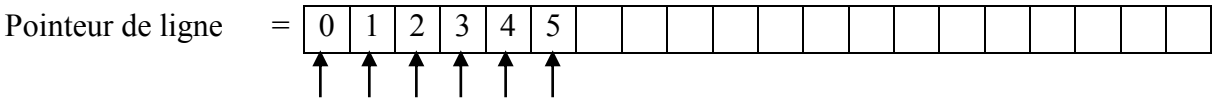
Il se peut que le nombre de lignes ne soit pas un multiple de la taille d'une chaîne. La dernière chaîne sera donc incomplète. Nous définissons donc *rem*, le nombre de fils de la dernière chaîne, à l'équation (3.7).

$$rem = M - \big(S_w \times (W - 1)\big). \tag{3.7}$$

Au début du programme, chaque fil doit accéder au vecteur de pointeurs de ligne pour connaître le début et la fin de sa ligne aux lignes 6 et 7. Deux instructions sont utilisées, nous diviserons donc T_{ptr} en la somme de T_{ptr1} et T_{ptr2}.

À la ligne 6 du pseudocode, il y a une première requête à la mémoire. Chaque fil accède à un élément consécutif du vecteur de pointeur de ligne, par exemple, le fil 0 doit lire l'élément à l'adresse 0 et le dernier fil de la chaîne doit lire l'élément à l'adresse S_w. Ceci correspond donc à accéder à la mémoire de façon séquentielle et alignée, soit un transfert coalescent.

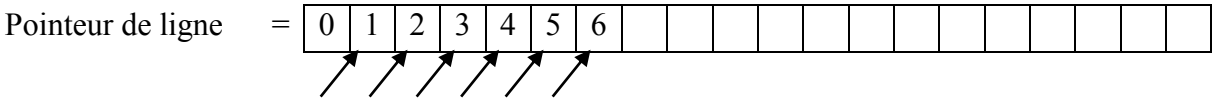
Dans l'exemple ci-dessous, nous supposons qu'une chaîne est constituée de six fils d'exécution. Chacun de ceux-ci accède à un élément du vecteur (représenté par les flèches).



L'exécution de la ligne 6 par une chaîne demande alors un nombre de transactions à la mémoire correspondant à l'équation (3.8) où T_{ptrl} représente le nombre de transactions qui est effectué suite à une demande de transfert d'éléments séquentiels et dont la première adresse est alignée. Le nombre d'éléments demandé N_e dépend de chaque chaîne, il est donc représenté comme un tableau dans l'équation.

$$T_{ptr1} = \sum_{i=0}^{W-1} \left\lceil \frac{N_e[i]}{S_{tx}/S_e} \right\rceil \tag{3.8}$$

À la ligne suivante (ligne 7), il y a une nouvelle requête au vecteur de pointeur de ligne. Chaque fil accède à des éléments dans un ordre séquentiel, mais à une adresse différente de son identifiant. Par exemple, le fil x a besoin de l'élément x+1 du vecteur, comme présenté ci-dessous.



Un transfert s'effectue toujours à partir de 0 ou à un multiple de la taille du transfert à la cache. Ainsi, la lecture demandée à la ligne 7 peut représenter un nombre de transactions plus élevé, même si chaque fil demande le même nombre d'adresses qu'à la requête précédente. Le nombre de transactions par chaîne pour cet accès séquentiel non-aligné correspond à l'équation (3.9). Ici, le nombre d'éléments manquant à l'alignement équivaut à 1.

$$T_{ptr2} = \sum_{i=0}^{W-1} \left\lceil \frac{N_e[i] + 1}{S_{tx}/S_e} \right\rceil \tag{3.9}$$

On pourrait penser que certaines simplifications sont possibles lorsque deux chaînes consécutives effectuent l'accès à la mémoire. La deuxième chaîne pourrait retrouver les éléments dont elle a besoin dans la cache. Par contre, comme expliqué plus tôt dans la méthodologie, pour un GPU, ceci n'est pas toujours le cas. Il est impossible de prédire à quel moment une chaîne en particulier

effectuera une instruction en particulier. Il est possible, par exemple, que lorsque la chaîne 0 effectue l'instruction 7, la prochaine chaîne puisse être la chaîne 20 et l'instruction à laquelle elle est arrivée est l'instruction de la ligne 12. Ainsi, le calcul du nombre de transactions ne prendra aucun compte de la cache.

Pour la plupart des chaînes, le nombre d'éléments demandé, N_e, est égal à un élément par fil d'exécution, donc S_w éléments. Par contre, pour la dernière chaîne, qui peut être incomplète, le nombre d'éléments demandé est égal à *rem*. Ceci est surtout important pour la ligne 7, car cela fait en sorte que le désalignement puisse être ignoré. Le nombre total de transactions à la mémoire pour le vecteur de pointeur de ligne est de :

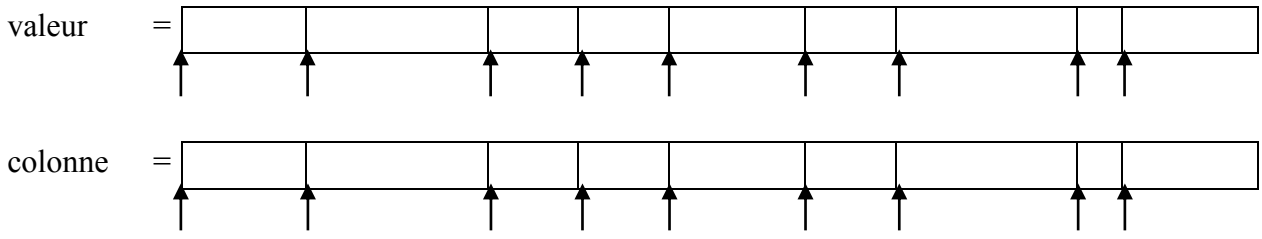
$$T_{ptr} = (W-1)\left(\left\lceil\frac{S_w}{S_{tx}/S_e}\right\rceil + \left\lceil\frac{S_w+1}{S_{tx}/S_e}\right\rceil\right) + \left\lceil\frac{rem}{S_{tx}/S_e}\right\rceil + \left\lceil\frac{rem+1}{S_{tx}/S_e}\right\rceil \qquad (3.10)$$

Les lignes 6 et 7 représentent les accès à la mémoire pour le vecteur de pointeur de ligne. Ainsi, le nombre d'instructions de requête I_R est égal à deux. En appliquant l'équation (3.1), nous avons :

$$R_{ptr} = 2W \qquad (3.11)$$

À la ligne 10, chaque fil doit multiplier une valeur de la matrice A, contenue dans le vecteur de valeurs, avec un élément du vecteur *x*, dont l'emplacement est donné par le vecteur de colonnes, et ce pour toute la ligne qui lui est assignée. Cette ligne de code se divise en 3 requêtes à la mémoire, un accès au vecteur de valeurs, un accès au vecteur de colonnes et un accès au vecteur *x*.

Chaque ligne de la matrice ne contient généralement pas le même nombre d'éléments non nuls. Les vecteurs de valeurs et de colonnes peuvent être représentés comme suit, chaque rectangle représentant une section ayant les éléments non nuls d'une même ligne.



Puisque les fils sont exécutés en groupe, une première itération est représentée avec les flèches où chaque fil accède au premier élément de sa ligne, ainsi que le numéro de colonne correspondant.

Les éléments demandés par ces fils sont rarement consécutifs, résultants donc à des transferts de données inutiles. De plus, les transferts ne sont pas toujours alignés. Si les lignes sont courtes, certains éléments pourront être intégrés à un même transfert de mémoire. Si une ligne est plus longue qu'un transfert, il n'est pas nécessaire de la transférer au complet.

Pour mesurer le nombre de transactions à la mémoire associé à la lecture de ces vecteurs, nous avons besoin du nombre d'éléments par lignes disponible grâce au tableau *Srow*. À l'aide du nombre d'éléments de chaque ligne de la matrice, il est possible de connaître la région d'adresses à laquelle chaque fil d'une même chaîne aura à accéder.

La Figure 3.3 présente l'algorithme complet qui est utilisé pour le calcul des nombres de transactions à la mémoire T_v, T_c et T_x correspondant à la demande d'accès en mémoire de l'instruction de la ligne 10 de la Figure 3.2.

```
 1   adresse = 0
 2   for i = 0 to W-1 do
 3
 4     Initialiser idxValChaine et idxColChaine à vide
 5     for j = 0 to Sw-1 do
 6       ligne = j + (i * Sw)
 7       if ligne ≤ M then
 8         for k = 0 to Srow[ligne]-1 do
 9           idxValChaine [j, k] = adresse
10           idxColChaine [j, k] = col[adresse]
11           adresse ++
12         end for
13       end if
14     end for
15
16     Tv += calcul_T(idxValChaine, maxRow, Sw)
17     Trier idxColChaine en ordre croissant
18     Tx += calcul_T(idxColChaine, maxRow, Sw)
19
20   end for
21   Tc = Tv
```

Figure 3.3 : Calcul du nombre de transactions à la mémoire des vecteurs de valeurs et de colonnes ainsi que du vecteur *x* pour le format CSR avec un fil d'exécution par ligne

Un tableau, appelé *idxValChaine*, est utilisé pour conserver les adresses de chaque élément non nul par ligne devant être accédé par une chaîne en particulier. Le tableau a S_w lignes et *maxRow* co-

lonnes. Chaque colonne de ce tableau représente les adresses d'une requête de la chaîne en question. Par exemple, la première colonne représente les adresses dont chaque fil a besoin pour avoir les informations de la valeur se trouvant au début de sa ligne, où *i* est égal à *debut* (ligne 9 à la Figure 3.2). La colonne suivante correspond aux adresses du deuxième élément non nul de la ligne associé aux fils.

Pour ce qui est du vecteur *x*, le nombre d'accès à la mémoire dépend principalement de la distribution des éléments non nuls de la matrice. Dans le meilleur des cas, tous les fils ont un élément non nul de la matrice se trouvant à la même colonne, ce qui correspondra au transfert d'un seul élément du vecteur *x*. Dans le pire des cas, les colonnes seront très dispersées et chaque fil aura besoin d'une transaction complète pour chercher un élément du vecteur *x*; soit S_w transactions par chaîne.

De la même façon que pour les vecteurs de valeurs et de colonnes, un tableau de taille identique appelé *idxColChaine* est instancié (voir Figure 3.3). Ce tableau contient les adresses à accéder dans le vecteur *x* pour chaque fil d'une même chaîne. Ces adresses dépendent des éléments du vecteur de colonnes.

Lorsque les tableaux *idxValChaine* et *idxColChaine* sont remplis, il suffit de compter le nombre de régions alignées qui devront être accédées pour récupérer ces adresses. La fonction *calcul_T()*, présentée à la Figure 3.4, sera utilisée où la variable *multiple* représente un multiple de l'adresse d'alignement, commençant à 0.

```
1   T calcul_T(tableau, n_lignes, n_colonnes)
2
3     for i = 0 to n_lignes-1 do
4       multiple = 0
5       for j = 0 to n_colonnes-1 do
6         if tableau[i,j] ≠ vide && ⌊tableau[i, j]⌋/(S_tx/S_e) ≥ multiple then
7           T ++
8           multiple = ⌊tableau[k, j]⌋/(S_tx/S_e) + 1
9         end if
10      end for
11    end for
12    Retourner T
```

Figure 3.4 : Fonction calcul_t() prenant en paramètre un tableau d'adresses et retournant le nombre de transactions à la mémoire que ces adresses effectuent

Une chaîne exécutant la ligne 10 du code de la Figure 3.2 demande trois requêtes à la mémoire à chaque itération de la boucle; une pour le vecteur de valeurs, une pour le vecteur de colonnes et une pour le vecteur *x*. Le nombre total de requêtes dépend du nombre d'éléments non nuls dans la plus longue ligne faisant partie de l'ensemble des lignes gérées par une chaîne. L'équation (3.12) permet de calculer ce résultat. L'équation $(1 + ((i - 1) \times S_w)$ représente la première ligne de la chaîne, puis le maximum de chaque chaîne est additionné.

$$R_v = R_c = R_x = \sum_{i=0}^{W-1} \max \begin{pmatrix} S_{row}[0 + (i \times S_w)], \\ S_{row}[1 + (i \times S_w)], \\ \dots \\ S_{row}[S_w - 1 + (i \times S_w)] \end{pmatrix} \tag{3.12}$$

Pour le vecteur *y*, accédé à la ligne 12 de la Figure 3.2, chaque fil inscrit le résultat du produit scalaire auquel il est assigné. Puisque chaque fil est assigné à une ligne de façon ordonnée, soit fil 0 pour la ligne 0 et fil $M - 1$ pour la dernière ligne $M - 1$, les résultats écrits dans le vecteur *y* sont disposés de manière séquentielle et alignée. Au total, nous avons un nombre de transactions et un nombre de requêtes calculé par les équations (3.13) et (3.14).

$$T_y = (W - 1) \left\lceil \frac{S_w}{S_{tx}/S_e} \right\rceil + \left\lceil \frac{rem}{S_{tx}/S_e} \right\rceil \tag{3.13}$$

$$R_y = W \tag{3.14}$$

3.2.2 Une chaîne de fils d'exécution par ligne

Une SpMV avec une matrice en format CSR peut aussi être implémentée d'une autre façon. Nous avons vu, avec la distribution des tâches précédente sur les fils d'exécution, que la lecture des vecteurs de valeur et de colonne était très désordonnée. Afin de remédier à ce problème, une solution est d'assigner une chaîne de fil d'exécution au calcul du produit scalaire d'une ligne de la matrice avec le vecteur *x*. Le code du noyau associé à cette distribution des tâches et inspiré du noyau nommé *vector kernel* [18] est présenté à la Figure 3.5.

Associer chaque chaîne à une ligne de la matrice permet aux fils d'une même chaîne d'accéder à des valeurs consécutives des vecteurs de valeurs et de colonnes, résultant à des accès à la mémoire

moins dispersés. Ceci fait aussi en sorte qu'autant de chaînes que de lignes sont nécessaires à l'exécution de ce noyau (3.15).

$$W = M \tag{3.15}$$

Chaque fil multiplie un élément de la ligne associé à sa chaîne avec l'élément du vecteur *x* et place son résultat dans un tableau résidant dans la mémoire partagée. Le deuxième élément de chaque fil se trouve S_w éléments plus loin, si la longueur de la ligne le permet.

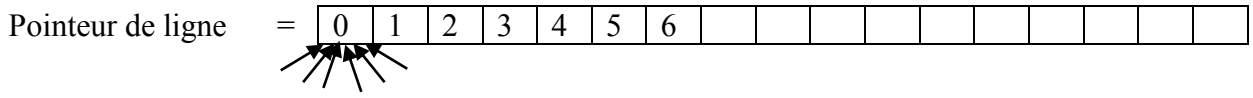
```
 1   extern __shared__ float sommes[];
 2   int indice_unique = blockIdx.x * blockDim.x + threadIdx.x;
 3   int ligne = indice_unique / 32;
 4   int indice_par_chaine = indice_unique & (32 -1);
 5
 6   sommes[threadIdx.x] = 0.0;
 7   int debut, fin;
 8
 9   if (ligne < M)
10   {
11      debut = ptr[ligne];
12      fin = ptr[ligne + 1];
13
14      for (int i = debut + indice_par_chaine; i < fin; i+=32)
15         sommes[threadIdx.x] += val[i] * x[col[i]];
16
17      // addition parallèle du tableau des sommes pour la chaîne en cours
18      if (indice_par_chaine < 16) sommes[threadIdx.x] += sommes[threadIdx.x + 16];
19      if (indice_par_chaine < 8) sommes[threadIdx.x] += sommes[threadIdx.x + 8];
20      if (indice_par_chaine < 4) sommes[threadIdx.x] += sommes[threadIdx.x + 4];
21      if (indice_par_chaine < 2) sommes[threadIdx.x] += sommes[threadIdx.x + 2];
22      if (indice_par_chaine < 1) sommes[threadIdx.x] += sommes[threadIdx.x + 1];
23
24      if(indice_par_chaine == 0)
25         y[ligne] = sommes[threadIdx.x];
26   }
```

Figure 3.5 : Code du noyau effectuant une SpMV utilisant le format CSR et attribuant une chaîne par ligne de matrice

La mémoire partagée est utilisée pour entreposer les différentes sommes calculées par les fils d'un même bloc. Les fils d'un bloc s'occupent de plusieurs lignes consécutives dont le nombre est connu par S_b / S_w. La variable *threadIdx.x*, utilisée pour lire dans la mémoire partagée, représente l'indice d'un fil à l'intérieur d'un bloc.

La prochaine étape consiste à additionner les résultats temporaires pour enfin stocker le résultat final dans le vecteur *y*. Dans le noyau présenté, l'addition de ces résultats temporaires se fait de façon parallèle. Le nombre de fils par chaîne doit être connu à l'avance pour dérouler la boucle permettant de faire une addition parallèle du tableau des sommes. Puisque tous les GPU NVIDIA ont 32 fils par chaîne, c'est donc ce qui est utilisé dans le code au lieu de la variable S_w. De plus, puisque les fils d'une même chaîne exécutent la même instruction au même moment, le programme n'a pas besoin d'utiliser de barrières de synchronisation. Dans le modèle, les accès à la mémoire partagée seront ignorés, car le temps requis pour ces accès est du même ordre que celui requis pour des accès à des registres (section 2.4.2.3).

Au début du programme, il faut lire le vecteur de pointeur de ligne. Chaque fil doit accéder à la mémoire pour connaître le début et la fin de la ligne à laquelle sa chaîne est assignée. Puisqu'une chaîne est associée à une ligne, tous les fils d'une même chaîne accèdent à une même adresse aux lignes 11 et 12. Par exemple, pour la ligne 0 et l'instruction à la ligne 11, nous avons :



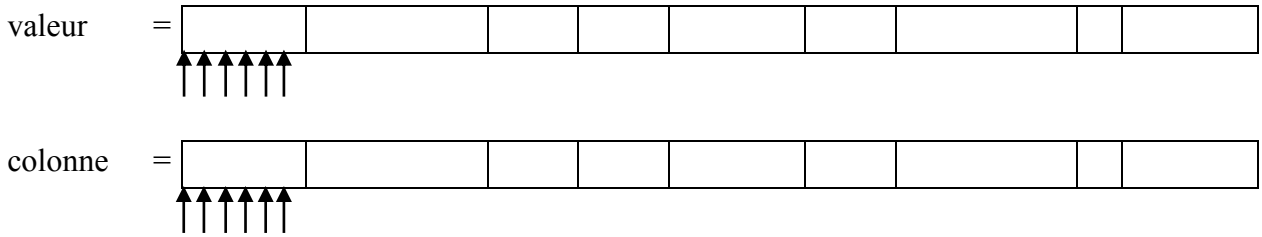
Si le nombre d'éléments transférables à la cache est très grand, n'ayant besoin que d'un élément par requête, plusieurs données inutiles sont transférées. Les équations (3.16) et (3.17) calculent le nombre de transactions et de requêtes de l'accès au vecteur de pointeur de ligne. La taille d'un élément à lire sera toujours plus petite que la taille du transfert. Ainsi, le résultat de la division de la taille de transfert par la taille de l'élément (S_{tx} / S_e) est un nombre entier positif égal à 1 ou plus. Ainsi, le résultat de la fonction plafond donnera toujours 1.

$$T_{ptr} = 2W \left\lceil \frac{1}{S_{tx}/S_e} \right\rceil = 2W \tag{3.16}$$

$$R_{ptr} = 2W \tag{3.17}$$

Pour les vecteurs de valeurs et de colonnes, chaque chaîne doit accéder à tous les éléments non nuls de la ligne qui lui est assignée. Ainsi, à la première itération, la chaîne tentera de demander l'adresse d'au plus les S_w premiers éléments de sa ligne, dépendamment du nombre d'éléments non nuls de la ligne. Nous avons donc des adresses séquentielles, mais pas toujours alignées. L'alignement dépend de la taille des lignes qui varie et qui n'est pas un multiple du nombre d'éléments

pouvant être transférés à la fois. Par exemple, si chaque chaîne est composée de 6 fils, la chaîne en question nécessiterait un accès à six valeurs non nulles ainsi que les colonnes associées à celles-ci.




```
1   for i = 0 to W-1 do
2     if Srow[i] ≠ 0 then
3       for j = 0 to Srow[i]-1, j += Sw do
4         a = (ptr[i] + j) modulo (Stx / Se)
5         Ne = Srow[i] – j
6         Tv+= ⌈(min(Ne, Sw) + a)/(Stx/Se)⌉
7       end for
8     end if
9   end for
10  Tc = Tv
```

Figure 3.6 : Calcul des nombres de transactions à la mémoire causés par les vecteurs de valeurs et de colonnes de l'implémentation utilisant le format CSR attribuant une chaîne par ligne

```
1   Tx = 0
2   for i = 0 to W-1 do
3     debut = ptr[i]
4     fin = ptr[i + 1]
5     for k = 0 to Srow[i]-1, k += Sw do
6
7       Initialiser idxColChaine à vide
8       for j = 0 to Sw-1 do
9         colIdx = debut + j + k
10        if (colIdx < fin) then
11          idxColChaine[j] = col[colIdx]
12        end if
13      end for
14
15      Trier idxColChaine en ordre croissant
16      Tx += calcul_T(idxColChaine, 1, Sw)
17
18    end for
19  end for
```

Figure 3.7 : Calcul du nombre de transactions à la mémoire causé par le vecteur *x* de l'implémentation utilisant le format CSR attribuant une chaîne par ligne

L'algorithme pour calculer le nombre de transactions pour les vecteurs de valeurs T_v et de colonnes T_c est présenté à la Figure 3.6. L'idée est d'effectuer une boucle sur toutes les chaînes et de mesurer le nombre d'éléments *a* manquant à l'alignement engendré par l'adresse du premier élément non nul de la ligne en cours. L'adresse du premier élément de chaque ligne est disponible grâce au vecteur *ptr* du format CSR. Le nombre de transactions est mesuré à chaque ligne et pour chaque itération des S_w fils nécessaires pour accéder à tous les éléments non nuls de la ligne. Le nombre de transactions pour le vecteur de colonnes est le même.

Le nombre de transactions au vecteur *x*, T_x, est calculé de la même façon qu'à la section précédente, à la Figure 3.3, puisqu'il dépend uniquement des éléments accédés dans le vecteur de colonnes. L'algorithme est présenté à la Figure 3.7.

Pour le nombre de requêtes, le nombre est le même pour les vecteurs de valeurs, de colonnes et *x*. Il est mesuré à l'aide du nombre de sections de S_w éléments non nuls par ligne. Dans l'équation (3.18), nous présentons le nombre de requêtes pour les vecteurs de valeurs de colonnes et du vecteur *x*. Le nombre total de requêtes pour ces trois vecteurs se calcule en effectuant une somme du nombre de requêtes de chaque chaîne. La variable *Srow*, représentant le nombre d'éléments par ligne de la matrice, est représentée par un tableau qui est aussi de la taille du nombre de chaînes en cours.

$$R_v = R_c = R_x = \sum_{i=0}^{W-1} \left\lceil \frac{Srow[i]}{S_w} \right\rceil \tag{3.18}$$

Enfin, le résultat est placé dans le vecteur *y* par le premier fil de chaque chaîne. Ainsi, il y a une requête par chaîne pour placer un élément en mémoire (3.19) (3.20). Encore une fois, la taille d'un élément est plus petite que la taille de transfert et le résultat de la fonction plafond équivaut toujours à 1.

$$T_y = W \left\lceil \frac{1}{S_{tx}/S_e} \right\rceil = W \tag{3.19}$$

$$R_y = W \tag{3.20}$$

3.3 Format ELLPACK

Dans cette section, nous présenterons le calcul du nombre de transactions et de requêtes de l'implémentation d'une SpMV avec une matrice en format ELLPACK. Comme expliqué à la section 2.2.4, le format ELL utilise deux matrices pour représenter la matrice creuse; une matrice contenant les valeurs des éléments, appelée *data*, et une matrice indiquant les indices de colonnes, appelée *indices*. Les matrices sont de taille *M* par *K* où *K* est égal au nombre d'éléments non nuls dans la plus longue ligne. Il se peut qu'il existe donc des éléments nuls dans ces matrices.

Les nombres de transactions et de requêtes de cette section seront séparés pour chaque vecteur en mémoire comme présenté à l'équation (3.21). Le nombre de transactions associé aux matrices de valeurs, T_d, et de colonnes, T_i, au vecteur x, T_x, et au vecteur y, T_y.

$$T_{ELL} = T_d + T_i + T_x + T_y \tag{3.21}$$

Le noyau de cette implémentation est présenté à la Figure 3.8. Il est inspiré du noyau nommé *spmv_ell_kernel* [18].

```
 1   int ligne = blockIdx.x * blockDim.x + threadIdx.x;
 2   float somme = 0.0;
 3   float v;
 4   int indice;
 5
 6   if (ligne < M)
 7   {
 8      for(int i = 0; i < K; i++)
 9      {
10         v = data[M * i + ligne];
11         if (v != 0.0)
12            somme += v * x[indices[M * i + ligne]];
13      }
14      y[ligne] = somme;
15   }
```

Figure 3.8 : Code du noyau effectuant une SpMV utilisant le format ELLPACK

Afin d'effectuer une multiplication entre une matrice en format ELL et un vecteur en parallèle, nous attribuons un fil pour chaque ligne. Chaque fil doit lire la valeur et la colonne des éléments de sa ligne. Après une opération de multiplication et accumulation, chaque fil place le résultat final dans le vecteur *y*. Nous avons donc un nombre de chaînes calculé par (3.6) et un nombre de fils

restants dans la dernière chaîne calculé par (3.7). Pour réduire le nombre d'accès à la mémoire, seule la matrice contenant les valeurs est accédée d'abord. Si l'élément lu est non nul, les autres accès sont demandés. Dans le modèle, nous mesurons les requêtes et les transactions à la mémoire des matrices *data* et *indices* et des vecteurs *x* et *y*.

À la première itération de la boucle *for*, chaque fil doit accéder au premier élément de la ligne à laquelle il est associé. Aux itérations suivantes, chaque fil demandera un élément suivant dans la ligne jusqu'à un total de *K* itérations. À la Figure 3.9, les flèches pointent aux adresses demandées par les fils de la première chaîne. Pour avoir un accès à des adresses séquentielles, il faut que les matrices *data* et *indices* soit misent en mémoire une colonne après l'autre; en ordre *column-major*.

Le nombre de requêtes nécessaires pour accéder à la matrice *data* est proportionnel au nombre de chaînes *W* et au nombre d'itérations *K*, comme le démontre l'équation (3.22).

$$R_d = WK \tag{3.22}$$

Pour le nombre de transactions, T_d, il faut savoir que chaque fil d'une même chaîne accède à un élément consécutif de la matrice. Par contre, les accès peuvent ne pas être alignés à cause du nombre de lignes *M* qui n'est pas nécessairement un multiple de la taille du transfert. Le nombre d'éléments *a* manquant à l'alignement est calculé pour chaque chaîne en prenant compte de l'adresse du premier élément de chaque colonne. L'algorithme est montré à la Figure 3.10.

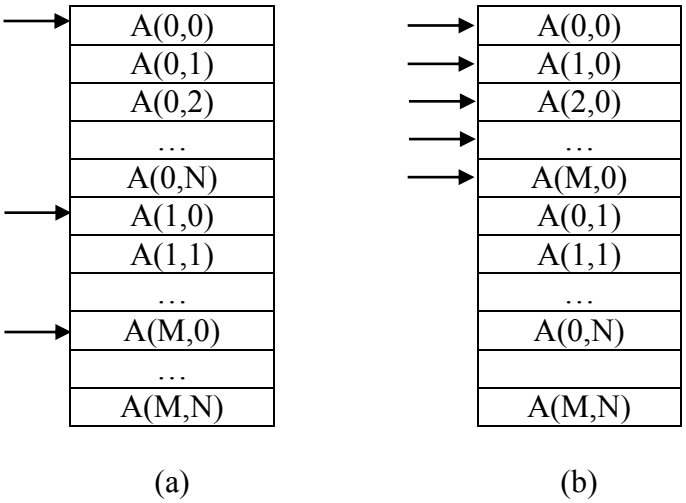


Figure 3.9 : Une matrice A représentée dans les deux ordres possibles a) *row-major* b) *column-major*

La matrice *indices* et le vecteur *x* sont accédés seulement lorsque la valeur lue dans la matrice *data* est non nulle. Puisque les matrices *data* et *indices* ont leurs éléments non nuls aux mêmes endroits,

l'emplacement des éléments invalides de la matrice *indices* peut être utilisé pour mesurer le nombre de requêtes R_i et R_x et le nombre de transactions à la mémoire T_i et T_x. La matrice des éléments composant la matrice de colonnes *indices* est créée à partir des vecteurs *row* et *col* de la représentation COO de la matrice creuse. Les vecteurs *row* et *col* sont accessibles suite à la lecture de la matrice à partir du fichier duquel elle est accédée.

```
1   for i = 0 to K-1 do
2     for j = 0 to W-1 do
3       a = (i * M + j * Sw) modulo (Stx / Se)
4       if j == W then
5         Td+= ⌈(rem + a)/(Stx/Se)⌉
6       else
7         Td+= ⌈(Sw + a)/(Stx/Se)⌉
8       end if
9     end for
10  end for
```

Figure 3.10 : Calcul des transactions à la mémoire de la matrice *data* du format ELL

```
1   for i = 0 to K-1 do
2     for j = 0 to W-1 do
3       for k = 0 to Sw-1 do
4         ligne = j * Sw + k
5         if ligne ≤ M && indices[ligne, i] ≠ invalide then
6           Ri ++
7           Sortir de la dernière boucle for
8         end if
9       end for
10    end for
11  end for
12  Rx = Ri
```

Figure 3.11 : Calcul des nombres de requêtes à la mémoire causés par la matrice *indices* et le vecteur *x* pour l'implémentation utilisant le format ELLPACK

Le nombre de requêtes nécessaire pour accéder à la matrice *indices* et au vecteur *x* est le même. Les instructions de lecture des deux données dépendent d'une même condition. Le nombre de requêtes est calculé à l'aide de trois boucles qui parcourent chaque colonne, chaque chaîne et chaque fil de la matrice *indices*. L'algorithme vérifie si un élément accédé par la chaîne en question est non nul et incrémente le nombre de requêtes si c'est le cas. L'algorithme passe ensuite à la prochaine chaîne (Figure 3.11).

Le nombre de transactions dues à la matrice *indices* est calculé à l'aide des mêmes trois boucles. Elles servent à identifier la longueur de la section à accéder; entre le premier élément non nul et le dernier. Cette valeur est ensuite additionnée au nombre manquant à l'alignement *a* pour en connaître le nombre de transactions nécessaires. Les détails se trouvent à la Figure 3.12.

```
 1   for i = 0 to K-1 do
 2     for j = 0 to W-1 do
 3       dataok = faux
 4       premier = dernier = 0
 5       for k = 0 to Sw-1 do
 6         ligne = j * Sw + k
 7         if ligne ≤ M && indices[ligne, i] ≠ invalide then
 8           if dataok est faux then
 9             premier = dernier = ligne
10             dataok = vrai
11           elseif dataok est vrai then
12             dernier = ligne
13           end if
14         end if
15       end for
16       if dernier ≠ 0 then
17         a = (i * M + (premier - 1)) modulo (Stx / Se)
18         Ti += ⌈(dernier − premier + 1 + a)/(Stx/Se)⌉
19       end if
20     end for
21   end for
```

Figure 3.12 : Calcul du nombre de transactions à la mémoire causé par la matrice *indices* du format ELL

```
 1   for i = 0 to K-1 do
 2     for j = 0 to W-1 do
 3       Initialiser idxColChaine à vide
 4       for k = 0 to Sw-1 do
 5         ligne = j * Sw + k
 6         if ligne <= M then
 7           idxColChaine[k] = indices[ligne, i]
 8         end if
 9       end for
22       Trier idxColChaine en ordre croissant
23       Tx += calcul_T(idxColChaine, 1, Sw)
10     end for
11   end for
```

Figure 3.13 : Calcul du nombre de transactions à la mémoire causé par le vecteur *x* pour l'implémentation utilisant le format ELLPACK

Pour le vecteur x, l'algorithme utilisé ressemble beaucoup à celui de la Figure 3.3 où il faut compter le nombre de régions d'adresses séquentielles et alignées. La différence est qu'il y a trois boucles au lieu de deux puisqu'il faut parcourir une matrice au lieu d'un vecteur. L'algorithme est présenté à la Figure 3.13.

Le stockage des résultats dans le vecteur y se fait une fois pour chaque fil. Nous avons donc des équations identiques à (3.13) et (3.14) pour le calcul respectif des transactions et des requêtes.

3.4 Format COO

Dans cette section, nous présenterons le calcul du nombre de transactions et de requête de l'implémentation de la SpMV avec une matrice en format COO. Le format COO a été présenté à la section 2.2.1. L'implémentation présentée dans cette section se base sur le code de la bibliothèque libre CUSP [14]. Les algorithmes analysés sont inspirés de trois noyaux nommés *spmv_coo_flat_kernel*, *spmv_coo_reduce_update_kernel* et *spmv_coo_serial_kernel* qui se trouvent à l'Annexe A.

Le premier noyau effectue la SpMV sur l'ensemble des éléments pouvant être contenus dans le plus grand nombre de chaînes complètes et se nomme *spmv_coo_flat_kernel*. Chaque chaîne dans le noyau s'occupe d'un nombre d'éléments non nuls choisi à l'avance à l'aide d'une variable nommée intervalle. Dans le modèle proposé, nous fixerons l'intervalle à S_w afin que chaque fil s'occupe d'un seul élément à la fois. Discuter de la pertinence d'avoir un intervalle de valeurs plus grand est considéré comme une amélioration future de ce projet. Chaque fil calcule le résultat de la multiplication entre un élément de la matrice et l'élément correspondant dans le vecteur. Il faut ensuite additionner tous les résultats de la multiplication faisant partie d'une même ligne.

Il se peut que les éléments non nuls d'une même ligne de la matrice soient pris en charge par des fils d'exécution faisant partie de différentes chaînes ou différents blocs. Lorsque les éléments d'une ligne sont pris en charge par deux blocs, il y a un problème, car ceux-ci ne peuvent être synchronisés. Le dernier fil de chaque chaîne met alors en mémoire, dans un vecteur temporaire, le résultat temporaire de la ligne à laquelle son élément non nul est associé. Il faut alors appeler un deuxième noyau, nommé *spmv_coo_reduce_update_kernel*, ayant un seul bloc pour effectuer les calculs restants. Enfin, le noyau appelé *spmv_coo_serial_kernel* effectue la SpMV sur les éléments restants, qui n'ont pas été traités dans les deux premiers noyaux, de façon séquentielle.

3.4.1 SpMV parallèle

Le nombre de transactions de cette section sera scindé en plusieurs parties comme le montre l'équation (3.23). Le nombre de transactions associé à un accès au vecteur de lignes, T_r, de colonnes, T_c, de valeurs, T_v, du vecteur x, T_x, et y, T_y. Puisque des écritures sont effectuées dans des vecteurs temporaires en vue de les utiliser dans le noyau suivant, un nombre de transactions est aussi appelé T_{tmp}. Le nombre de requêtes sera scindé de la même façon.

$$T_{COO-1} = T_r + T_c + T_v + T_x + T_y + T_{tmp} \tag{3.23}$$

Ce noyau est lancé en assignant un fil par éléments non nuls de la matrice. Par contre, chaque chaîne doit être complète. Nous avons donc un nombre de chaînes mesuré par l'équation (3.24) qui traite d'un nombre d'éléments non nuls mesuré par l'équation (3.25).

$$W_{flat} = \left\lfloor \frac{N_{NZ}}{S_w} \right\rfloor \tag{3.24}$$

$$N_{NZflat} = W_{flat} \times S_w \tag{3.25}$$

Au début du code, le dernier fil de chaque chaîne doit lire une valeur d'indice de ligne et la placer dans la mémoire partagée. Il y a deux accès au vecteur de lignes, nous séparerons T_r en T_{r1} et T_{r2} et R_r en R_{r1} et R_{r2}. Le nombre de transactions et le nombre de requêtes associés à cette action sont calculés grâce aux équations (3.26) et (3.27). Puisque la taille d'un élément est toujours plus petite que la taille de transfert, le résultat de la fonction plafond donne toujours 1.

$$T_{r1} = W_{flat} \left\lceil \frac{1}{S_{tx}/S_e} \right\rceil = W_{flat} \tag{3.26}$$

$$R_{r1} = W_{flat} \tag{3.27}$$

Les fils doivent ensuite accéder aux vecteurs d'indices de ligne, d'indices de colonne, et de valeurs. Chaque fil étant assigné à un élément non nul consécutif, les accès se font de façon coalescente avec un nombre de transactions donné par (3.28) et un nombre de requêtes donné par (3.29).

$$T_{r2} = T_c = T_v = W_{flat}\left\lceil \frac{S_w}{S_{tx}/S_e} \right\rceil \tag{3.28}$$

$$R_{r2} = R_c = R_v = R_x = W_{flat} \tag{3.29}$$

Chaque fil a ensuite besoin d'un élément du vecteur *x* accédé à l'adresse correspondant à l'indice de colonne. Le nombre de requêtes est le même que pour R_{r2} (3.29). Le nombre de transactions à la mémoire créé par cette action est mesuré de façon similaire aux autres formats et est présenté à la Figure 3.14. Une première boucle crée le vecteur *idxColChaine* avec les adresses qui devront être accédées par une chaîne. La fonction *calcul_T()*, présentée à la Figure 3.4, est ensuite utilisée pour parcourir le vecteur ainsi créé et calculer le nombre de sections consécutives et alignées de la mémoire qui devront être transférées

```
1  for i = 0 to Wflat-1 do
2     Initialiser idxColChaine à vide
3     for j = 0 to Sw-1 do
4        ligne = i * Sw + j
5        idxColChaine[j] = col[ligne]
6     end for
7     Trier idxColChaine en ordre croissant
8     Tx += calcul_T(idxColChaine, 1, Sw)
9  end for
```

Figure 3.14 : Calcul du nombre de transactions à la mémoire causé par le vecteur *x* pour l'implémentation utilisant le format COO

Le produit de l'élément non nul et de l'élément du vecteur *x* est stocké dans la mémoire partagée. Les résultats faisant partie d'une même ligne sont alors additionnés de façon parallèle par chaque chaîne. Si un fil de la chaîne se trouve à être aussi à la fin d'une ligne, le résultat temporaire est ajouté au vecteur *y*. Le nombre de transactions se calcule de façon similaire au vecteur *x*. La différence est que les transactions au vecteur *y* dépendent du vecteur de lignes *row*. De plus, il y a une requête à la mémoire seulement si deux indices de lignes consécutifs sont différents, d'où l'utilisation d'un tableau appelé *adrEcritureChaine* qui liste seulement les adresses auxquelles il y aura une écriture. Le nombre de transactions à la mémoire est mesuré par le code à la Figure 3.15 et le nombre de requêtes à la Figure 3.16.

Dans ce noyau, plusieurs fils d'exécutions peuvent écrire à la même adresse du vecteur *y*. Ainsi, ce qui est placé en mémoire est additionné au contenu précédent. Ceci fait en sorte que les nombres

de transactions et de requêtes à la mémoire du vecteur *y* sont des lectures et des écritures et devront être multipliés par deux.

```
1   for i = 0 to Wflat-1 do
2     Initialiser idxRowChaine à vide
3     for j = 0 to Sw-1 do
4       ligne = i * Sw + j
5       idxRowChaine[j] = row[ligne]
6     end for
7     Initialiser adrEcritureChaine à vide
8     for j = 0 to Sw-2 do
9       if idxRowChaine[j] ≠ idxRowChaine[j + 1] then
10        adrEcritureChaine[j] = idxRowChaine[j]
11      end if
12    end for
13    Ty += calcul_T(adrEcritureChaine, 1, Sw)
14  end for
```

Figure 3.15 : Calcul des transactions à la mémoire causées par le vecteur *y* pour le format COO

Enfin, à la fin du noyau, le dernier fil de la chaîne inscrit ses résultats temporaires dans des vecteurs appelés *temp_rows* et *temp_results* situés dans la mémoire globale. L'addition de ces éléments temporaires sera réalisée dans un autre noyau. Deux écritures sont effectuées par un seul fil par chaîne. Nous avons donc un nombre de transactions et de requêtes mesuré par (3.30) et (3.31).

```
1   for i = 0 to Wflat-1 do
2     Initialiser idxRowChaine à vide
3     for j = 0 to Sw-1 do
4       ligne = i * Sw + j
5       idxRowChaine[j] = row[ligne]
6     end for
7     for j = 0 to Sw-2 do
8       if idxRowChaine[j] ≠ idxRowChaine[j + 1] then
9         Ry ++
10      end if
11    end for
12  end for
```

Figure 3.16 : Calcul des requêtes à la mémoire causées par le vecteur *y* pour le format COO

$$T_{tmp} = 2W_{flat}\left\lceil \frac{1}{S_{tx}/S_e} \right\rceil = 2W_{flat} \tag{3.30}$$

$$R_{tmp} = 2W_{flat} \tag{3.31}$$

3.4.2 Addition des résultats temporaires

Le deuxième noyau permettant d'effectuer une SpMV sur une matrice en format COO réalise les dernières opérations sur les résultats temporaires du noyau précédent. Le noyau est lancé avec un seul bloc pour permettre la synchronisation entre tous les fils d'exécution.

Chaque élément des vecteurs temporaires est ajouté au vecteur final *y* à l'adresse correspondante. Dans un premier temps, l'algorithme lit les vecteurs temporaires *temp_rows* et *temp_results* et les place dans la mémoire partagée. Par la suite, chaque fil d'exécution vérifie s'il est en fin de ligne en lisant dans la mémoire partagée les valeurs de lignes lues précédemment. Si c'est le cas, il additionne, dans le vecteur *y*, à la ligne correspondante, le résultat temporaire placé au préalable en mémoire partagée. Pour calculer les transactions et requêtes à la mémoire de ce noyau, le modèle a besoin du contenu de ces vecteurs temporaires. Ils peuvent être créés en se basant sur l'algorithme du noyau *spmv_coo_flat_kernel* présenté en annexe. Le modèle suit de près la structure de l'algorithme de *spmv_coo_reduce_update_kernel*.

Le nombre de transactions de cette section sera scindé en plusieurs parties comme le montre l'équation (3.32). Le nombre de transactions associé à un accès aux vecteurs temporaires, T_{tmp}, et au vecteur *y*, T_y. Le nombre de requêtes sera scindé de la même façon.

$$T_{COO-2} = T_{tmp} + T_y \tag{3.32}$$

Une requête à la mémoire pour la lecture des deux vecteurs temporaires est enregistrée si au moins un fil par chaîne effectue l'instruction de lecture. Le nombre de transactions associé à chaque requête est mesuré de la même façon tout en prenant compte du nombre de fils effectuant l'instruction par le nombre d'éléments qu'il est possible de transférer à la fois. Nous pouvons résumer les équations utilisées par (3.33) et (3.34) en utilisant N_w pour représenter le nombre de chaînes effectuant l'instruction de lecture et N_t le nombre de fils effectuant la lecture pour chacune de ces chaînes.

$$T_{tmp} = 2\sum_{i=1}^{N_w} \left\lceil \frac{N_t[i]}{S_{tx}/S_e} \right\rceil \tag{3.33}$$

$$R_{tmp} = 2N_w \tag{3.34}$$

L'écriture du résultat dans le vecteur *y* est réalisée lorsqu'un fil d'exécution est en fin de ligne. Les adresses qui seront accédées lors de l'écriture proviennent du vecteur *temp_rows* et sont notées dans un vecteur appelé *blockRowIdx* ayant autant d'éléments que de nombre de fils d'exécution dans le bloc. Une boucle est utilisée pour calculer le nombre de sections d'adresses alignées et consécutives que cela nécessite. Pour les requêtes, on ne compte que le nombre de chaînes ayant un fil d'exécution sur une fin de ligne (3.35).

$$R_y = N_{wfin} \tag{3.35}$$

Les résultats sont ajoutés à ceux qui sont déjà en mémoire. Ainsi, chaque écriture au vecteur *y* est aussi une lecture. Il faut donc prendre compte des variables T_y et R_y lors du calcul du nombre total de lectures.

3.4.3 SpMV séquentielle

À la fin du traitement, un noyau est lancé afin de multiplier les éléments de la matrice qui avaient été laissés de côté. Un seul fil d'exécution est utilisé dans ce noyau, ce qui rend le calcul séquentiel. Le code est très simple et consiste en une boucle qui itère sur les éléments restants et qui effectue une opération de multiplication-accumulation (Figure 3.17).

```
1   for(int i = 0; i < rem; i++)
2   {
3       y[row[i]] += val[i] * x[col[i]];
4   }
```

Figure 3.17 : Code du noyau séquentiel de l'implémentation de la SpMV utilisant le format COO

Il y a une requête à la mémoire pour chaque vecteur et chaque élément non nul accédés. Le nombre de requêtes et de transactions est le même puisqu'il n'y a qu'un seul fil dans la chaîne. Les calculs sont présentés aux équations suivantes :

$$rem = N_{NZ} - \left(S_w \left(\left\lceil \frac{N_{NZ}}{S_w} \right\rceil - 1 \right) \right) \tag{3.36}$$

$$T_{loads} = R_{loads} = T_r + T_c + T_v + T_x + T_y = 5rem \tag{3.37}$$

$$T_{stores} = R_{stores} = T_y = rem \tag{3.38}$$

3.5 Format hybride ELL/COO

Dans cette section, nous calculerons les nombres de transactions et de requêtes à la mémoire effectués par une implémentation de la SpMV utilisant le format HYB, un format hybride entre les formats ELLPACK et COO. Ce format a été introduit par Bell et Garland [18] et il leur permettait d'avoir la meilleure performance possible pour leur ensemble de matrices. Les nombres de transactions, *T*, et de requêtes, *R*, seront scindés en plusieurs nombres de la même façon que pour les formats ELL et COO présentés aux deux sections précédentes.

Comme vu à la section 2.2.5, le format choisit une valeur de *K* idéale pour avoir une bonne proportion d'éléments non nuls en format ELLPACK. Le reste des valeurs non nulles sont stockées dans un format COO. Le calcul des requêtes et transactions à la mémoire sera donc le même que pour les deux sections précédentes. La différence est dans la valeur *K* pour ELLPACK et la valeur N_{NZ} pour COO.

```
1   K = maxRow
2   for i = 1 to maxRow do
3     n_row = 0
4     for j = 0 to M-1 do
5       if Srow[j] ≥ i then
6         n_row ++
7       end if
8     end for
9     if n_row < M/3 then
10      K = i - 1
11      Arrêter le programme
12    end if
13  end for
```

Figure 3.18 : Calcul du paramètre *K* optimal pour le format HYB

La valeur *K* optimale peut être mesurée comme à la Figure 3.18. Dans la librairie CUSP [14], d'autres paramètres sont pris en compte dans le calcul du *K*. Différentes valeurs pourraient être données à *K* afin de mesurer son impact sur une SpMV, ce sujet est une des améliorations possibles à ce projet de recherche. À l'aide du paramètre *K*, les valeurs de *M* et N_{NZ} pour la section COO du format hybride peuvent être mesurées. L'algorithme utilisé, présenté à la Figure 3.19, se base sur le nombre d'éléments par ligne de la matrice. Si le nombre dépasse la valeur *K*, le nombre d'éléments restant additionné à la valeur de N_{NZ} du format COO. De plus, le nombre de lignes du format COO est incrémenté.

```
1   cooN_NZ = 0
2   cooM = 0
3   for i = 0 to M-1 do
4     if Srow[i] > K then
5       cooM ++
6       cooN_NZ += Srow[i] - K
7     end if
8   end for
```

Figure 3.19 : Calcul du paramètre *M* et N_{NZ} de la partie COO du format HYB

La matrice *indices* et les vecteurs *row* et *column* du format COO peuvent être créés à partir des valeurs d'indices de ligne et de colonne de la matrice originale. La matrice *indices* est créée de la même façon qu'au format ELL à la différence d'une nouvelle condition qui vérifie qu'on ne dépasse pas *K* colonnes. Si cette condition est fausse, on écrit alors dans les vecteurs *row* et *column* du format COO.

Les nombres de transactions et de requêtes sont ensuite calculés en appelant les fonctions des formats ELLPACK et COO qui ont été présentées aux deux sections précédentes. Si *K* est égal à zéro, seule la fonction du format COO sera appelée. Si *K* est égal à *maxRow* seule la fonction du format ELL sera appelée.

CHAPITRE 4 VALIDATION DU MODÈLE ET RÉSULTATS

Le modèle proposé au Chapitre 3 a été validé en comparant le nombre estimé de transactions et de requêtes aux valeurs mesurées par le logiciel NVIDIA Visual Profiler. Dans le présent chapitre, nous présenterons d'abord la méthodologie utilisée pour les tests, puis les résultats obtenus. Nous passerons ensuite à la discussion où nous évaluerons la performance des SpMV à l'aide du modèle. Nous discuterons de la précision du modèle, de la comparaison entre les différents formats et de l'estimation du temps d'exécution. Nous verrons ensuite une comparaison des résultats de ce projet de recherche avec ceux de la littérature, puis le chapitre se terminera sur des améliorations possibles du modèle proposé.

4.1 Méthodologie

4.1.1 Matrices creuses

Les matrices utilisées proviennent de la collection de matrices creuses de l'Université de Floride [15] d'où elles peuvent être téléchargées en format Matrix Market [16]. Ces matrices sont présentées dans le Tableau 4.1 en ordre de nombre de lignes. Pour chaque matrice, le tableau donne son nom, une brève description, sa dimension, le nombre d'éléments non nuls N_{NZ} et la densité d. Pour les matrices carrées, on n'indique qu'une dimension.

Pour avoir des résultats uniformes et comparables entre eux, chaque matrice a été traitée comme si elle était composée de nombres à virgule flottante. Ainsi, la matrice *conf5_4-8x8-05*, étant originellement complexe, et les matrices *mc2depi* et *rail4284* étant représentées en nombres entiers, ont été modifiées pour que les valeurs de leurs éléments non nuls soient représentées avec des nombres à virgule flottante. Si le type des valeurs n'est pas le même pour toutes les matrices, il faut faire des modifications aux noyaux pour qu'ils puissent recevoir des vecteurs de types différents. Pour des nombres complexes, il faudrait indiquer au noyau qu'il reçoit un type différent de valeurs et lui demander de toujours accéder à deux valeurs qui se suivent. Dans ce cas, le modèle présenté au chapitre précédent n'est plus valide. Un accès coalescent avec des valeurs réelles ne l'est pas pour un accès à des valeurs complexes, car la demande d'accès à la partie réelle est une section de mémoire deux fois plus grande où une adresse sur deux est nécessaire. Utiliser des matrices ayant des éléments de types différents est une des pistes suggérées pour des travaux futurs liés à ce projet.

Tableau 4.1 : Matrices utilisées dans ce mémoire

#	Nom	Description	Dimension	N_{NZ}	d
1	Harvard500	Web matrix	500	2 636	1.05E-02
2	rail4284	Railways set cover Constraint matrix	4 284 × 1 096 894	11 284 032	2.40E-03
3	wiki-Vote	Wikipedia who-votes-on-whom network	8 297	103 689	1.51E-03
4	California	Pages matching the query "California"	9 664	16 150	1.73E-04
5	wb-cs-stanford	Stanford CS web	9 914	36 854	3.75E-04
6	linverse	matrix inverse approximation	11 999	95 977	6.67E-04
7	ex11	computational fluid dynamics problem	16 614	1 096 948	3.97E-03
8	OPF_6000	Power simulation matrix	29 902	274 697	3.07E-04
9	pdb1HYS	Protein data bank 1HYS	36 417	4 344 765	3.28E-03
10	OPF_10000	Power simulation matrix	43 887	426 898	2.22E-04
11	rma10	3D CFD model of Charleston harbor	46 835	2 329 092	1.09E-03
12	conf5_4-8x8-05	Quantum chromodynamics	49 152	1 916 928	7.93E-04
13	nasasrb	Structure from NASA	54 870	2 677 324	8.89E-04
14	cant	FEM cantilever	62 451	4 007 383	1.03E-03
15	finan512	Economic problem	74 752	596 992	1.07E-04
16	consph	FEM concentric spheres	83 334	6 010 480	8.65E-04
17	hcircuit	Motorola circuit simulation	105 676	513 072	4.59E-05
18	cop20k_A	Accelerator cavity design	121 192	2 624 331	1.79E-04
19	shipsec1	FEM Ship section/detail	140 874	3 568 176	1.80E-04
20	scircuit	Motorola circuit simulation	170 998	958 936	3.28E-05
21	mac_econ_fwd500	Macroeconomic model	206 500	1 273 389	2.99E-05
22	pwtk	Pressurized wind tunnel	217 918	11 524 432	2.43E-04
23	web-Stanford	Web graph of Stanford.edu	281 903	2 312 497	2.91E-05
24	cnr-2000	web crawl of Italian CNR domain	325 557	3 216 152	3.03E-05
25	amazon0505	Amazon product co-purchasing network from May 5 2003	410 236	3 356 824	1.99E-05
26	neos	Linear programming problem	479 119 × 515 905	1 526 794	6.18E-06
27	mc2depi	2D Markov model of epidemic	525 825	2 100 225	7.60E-06
28	flickr	2005 crawl of flickr.com	820 878	9 837 214	1.46E-05
29	web-Google	Web graph from Google	916 428	5 105 039	6.08E-06
30	webbase-1M	Web connectivity matrix	1 000 005	3 105 536	3.11E-06

Ces matrices ont été choisies, car elles sont étudiées par Williams et al. [30] et reprises par Bell et Garland [18] et Guo et al. [47]. Les changements réalisés sont cohérents avec les matrices utilisées par Williams si nous nous fions aux différences rapportées[1]. Williams utilise l'emplacement des valeurs non nulles des matrices qu'il utilise plutôt que de la valeur des éléments qui s'y trouvent.

La représentation Matrix Market permet aussi de stocker des matrices dans un format binaire où seulement l'emplacement des éléments non nuls est stocké. Dans le code, le vecteur de valeurs de ces matrices sera créé à partir de nombres aléatoires à virgule flottante. De la même façon, le vecteur x, avec lequel la matrice est multipliée, est un vecteur de nombres aléatoires à virgule flottante. Les indices de lignes et de colonnes sont des nombres entiers. Le paramètre S_e, représentant la taille des éléments en mémoire, est donc égal à quatre octets dans tous les cas.

4.1.2 Environnement du modèle

Le modèle a été implémenté avec MATLAB. Le calcul du nombre de transactions et de requêtes est réalisé par une fonction pour chaque format qui prend en paramètre le nom du fichier de la matrice creuse ciblée. La matrice, entreposée dans un fichier de type Matrix Market, est lue à l'aide de la fonction *mmread.m* fournie par Boisvert et al. [17]. La fonction de lecture retourne la matrice creuse lue, sa taille et son nombre d'éléments non nuls. Le modèle extrait ensuite les vecteurs *row* et *column* de la matrice creuse, qui sont les indices de lignes et de colonnes correspondant à l'interprétation dans le format COO (voir la section 2.2.1). Puisque MATLAB utilise une représentation des données en ordre de colonnes (*column-major*), il est important d'extraire les vecteurs de la matrice transposée comme le montre l'exemple de la Figure 4.1. De plus, il faut réviser la valeur de N_{NZ} dans le cas où la matrice serait stockée de façon symétrique et que seule la moitié des éléments non nuls serait inscrite dans le fichier.

```
1   At = transpose(A);
2   [col, row] = find(At);
3   NNZ = size(row, 1);
```

Figure 4.1 : Code MATLAB permettant d'extraire le nombre d'éléments non nuls et les vecteurs *row* et *column* du format COO d'une matrice creuse *A*

[1] Voir les détails des différences au https://www.cise.ufl.edu/research/sparse/mat/Williams/README.txt

Chaque fonction a ensuite accès aux paramètres S_w et S_{tx} propres au GPU utilisé. Les valeurs de ces paramètres sont présentées à la section suivante.

4.1.3 Environnement de test

Les tests ont été réalisés sur deux différents postes de travail disposant chacun d'une carte graphique. Le logiciel Nsight Eclipse Edition a été utilisé sous le système d'exploitation Linux. C'est un environnement de programmation permettant de créer des projets en langage CUDA C/C++. La version de CUDA utilisée est 6.0.

De la même façon que pour le modèle, le code CUDA prend en paramètres le nom d'un fichier de matrice creuse en format Matrix Market [16], puis la fonction de lecture fournie par Boisvert et al. [17] dans les fichiers *mmio.h* et *mmio.c* est utilisée pour créer les vecteurs utiles au format COO (section 2.2.1). Plusieurs routines en C/C++ ont été écrites pour faire la conversion vers les différents autres formats utilisés : CSR, ELLPACK et HYB (hybride ELL/COO).

Nous avons à notre disposition deux cartes graphiques de l'architecture Kepler; une GeForce GTX 670 et une Tesla K20c. La première est une carte accessible par tous, elle est d'ailleurs utilisée pour des activités personnelles telles que les jeux vidéo ou la manipulation de photos et de vidéos numériques. La deuxième, une carte Tesla, est conçue pour être utilisée dans des serveurs, pour des applications nécessitant un temps de traitement élevé. Un résumé de leurs caractéristiques se trouve au Tableau 4.2.

Tableau 4.2 : Caractéristiques des cartes GPU à notre disposition

Caractéristique	**GeForce GTX 670**	**Tesla K20c**
Nombre de multiprocesseurs	7	13
Nombre de cœurs CUDA	1344	2496
Fréquence d'horloge	0.98 GHz	0.71 GHz
Fréquence de la mémoire (GDDR5)	3004 MHz	2600 MHz
Largeur du bus de mémoire	256-bit	320-bit
Nombre de fils par chaîne (S_w)	32	32
Taille du transfert en octet (S_{tx})	128	128
Taille de la mémoire globale	2048 Mo	4800 Mo
Taille de la mémoire partagée	48 Ko	48 Ko
Taille de la cache L1	16 Ko	16 Ko
Taille de la cache L2	512 Ko	1280 Ko
Prix de lancement	399 USD	3199 USD

Le nombre de transactions et de requêtes à la mémoire est mesuré par le profileur intégré au logiciel Nsight nommé NVIDIA Visual Profiler. Ce dernier permet de profiler tous les noyaux lancés par une application. Les paramètres utiles dans le cadre de ce projet sont nommés « Global Load Transactions » et « Global Store Transactions » pour récupérer le nombre de transactions et « gld request » et « gst request » pour le nombre de requêtes.

4.2 Résultats

Dans cette section seront présentés les nombres de transactions et de requêtes estimés et mesurés pour chaque format. Les différences entre les résultats du modèle et du profilage se calculent de la façon suivante :

$$D_T = \frac{T_{total\ mesuré} - T_{total\ estimé}}{T_{total\ mesuré}} \tag{4.1}$$

La valeur de D_T représente la différence relative du nombre de transactions. Le même calcul peut être effectué avec le nombre de requêtes total mesuré et estimé permettant d'obtenir D_R, la différence relative du nombre de requêtes. Lorsque la valeur de D_T ou D_R est négative, cela indique que le modèle surestime le nombre de transactions ou de requêtes demandé.

Nous avons remarqué que le nombre de transactions et de requêtes est le même pour les deux GPU à notre disposition. Un seul résultat sera donc présenté pour les nombres mesurés par le profileur.

4.2.1 Format CSR

Une SpMV utilisant le format CSR peut être implémentée en parallèle de deux façons. Une ligne de la matrice peut être gérée par un fil d'exécution ou une chaîne de fils. Tous les résultats exacts du nombre de transactions et de requêtes du format CSR se retrouvent à l'Annexe B

4.2.1.1 Un fil d'exécution par ligne

Le nombre de transactions total estimé et mesuré est présenté à la Figure 4.2. Les matrices ont été placées en ordre croissant du nombre d'éléments non nuls, N_{NZ}. On remarque une tendance à une augmentation du nombre de transactions, puisque la taille de la matrice formatée dépend de N_{NZ}. La différence entre le nombre de transactions estimé et mesuré est présentée à la Figure 4.3. L'erreur moyenne des différences en valeur absolue est de 1.072 %. On remarque que les matrices

ayant moins de valeurs non nulles que la matrice *conf5_4-8x8-05* sont plus souvent mal estimées avec une moyenne d'erreur de 1.9 %, contrairement à 0.4 % pour les autres.

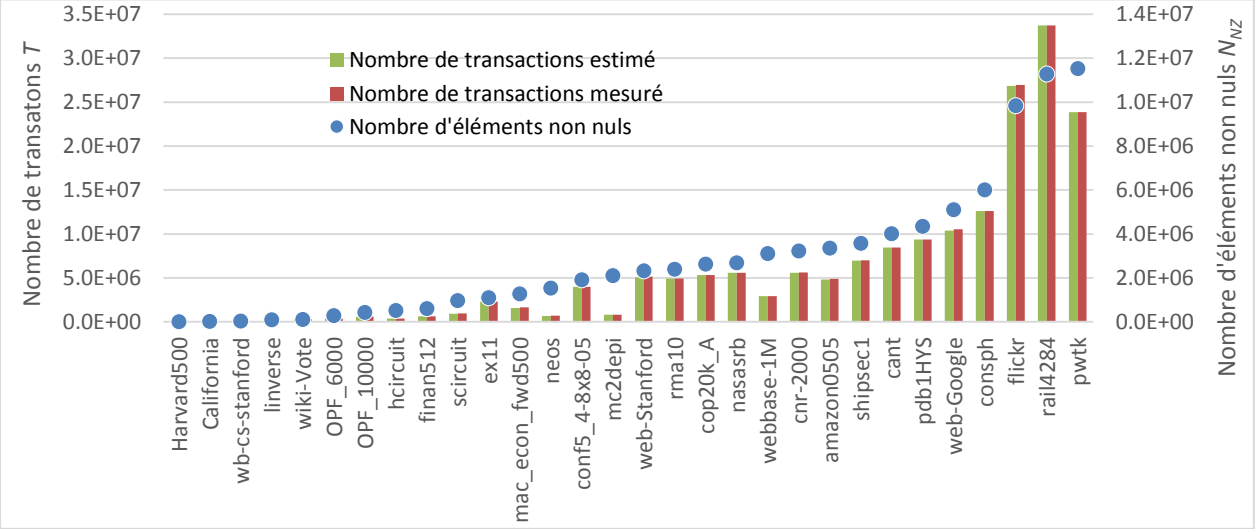


Figure 4.2 : Nombres de transactions estimés et mesurés pour le format CSR avec un fil par ligne comparés au nombre d'éléments non nuls de chaque matrice

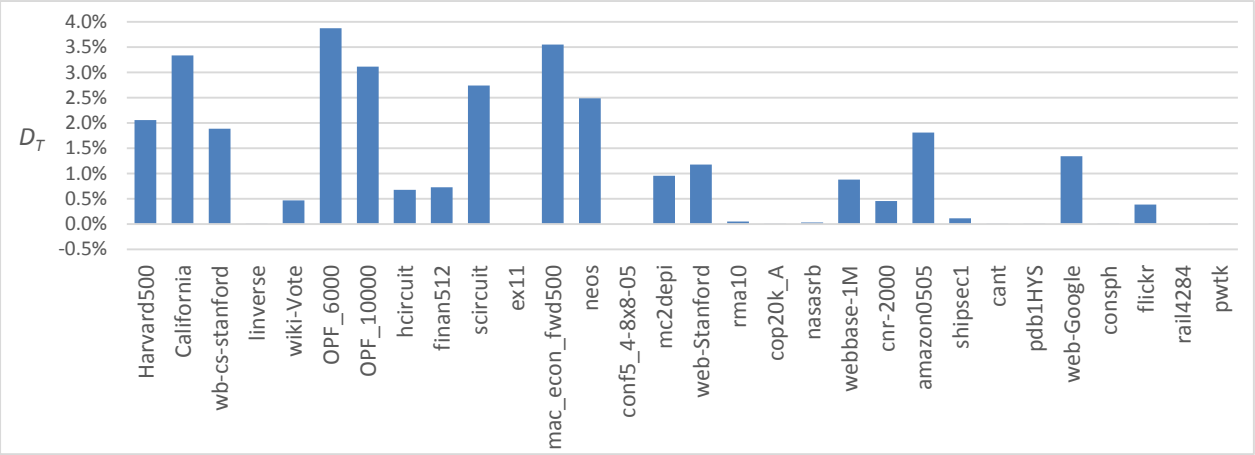


Figure 4.3 : Différences relatives entre le nombre de transactions estimé et mesuré pour le format CSR avec un fil par ligne

Les requêtes sont présentées à la Figure 4.4 avec les matrices placées dans le même ordre croissant du nombre d'éléments non nuls. La différence entre les valeurs estimées et mesurées se trouve à la Figure 4.5. L'erreur relative moyenne de cet ensemble est de 1.816%.

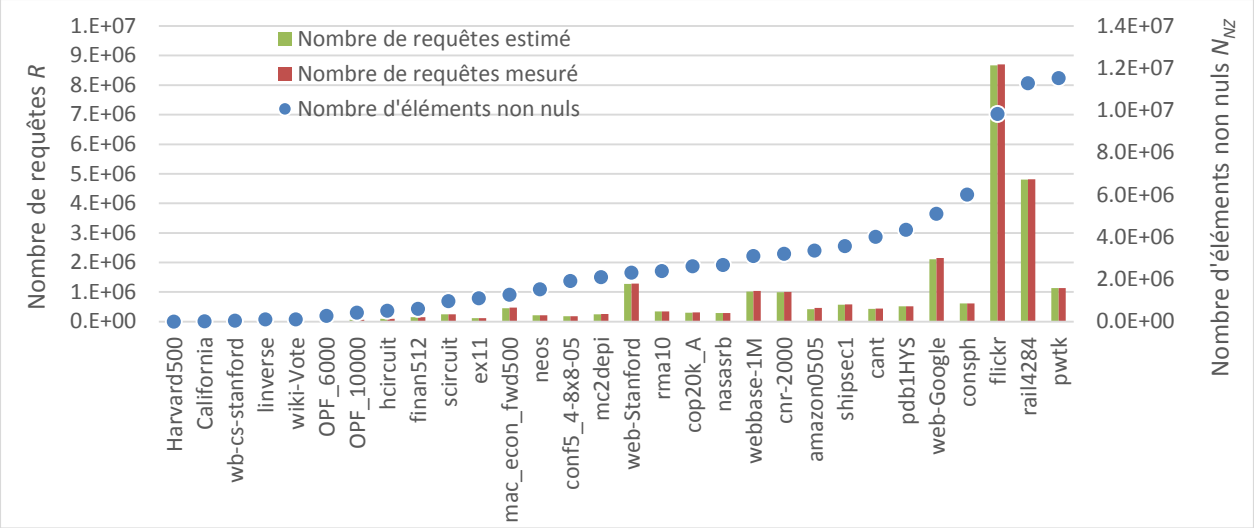


Figure 4.4 : Nombres de requêtes estimés et mesurés pour le format CSR avec un fil par ligne de la matrice comparés au nombre d'éléments non nuls de chaque matrice

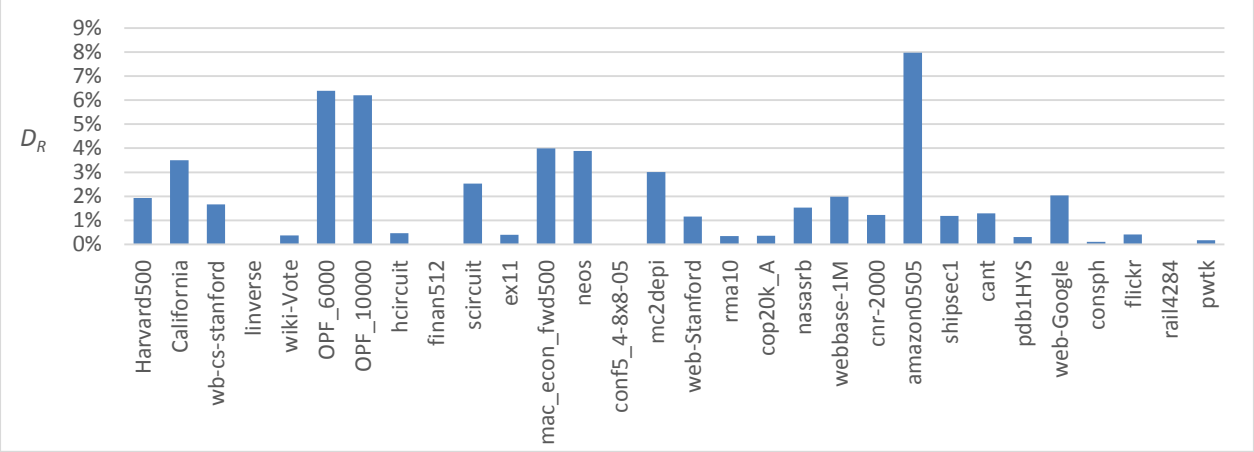


Figure 4.5 : Différences relatives entre le nombre de requêtes estimé et mesuré pour le format CSR avec un fil par ligne de la matrice

4.2.1.2 Une chaîne de fils d'exécution par ligne

Un graphique représentant les résultats du nombre de transactions estimé et mesuré pour le format CSR avec une chaîne par ligne est présenté la Figure 4.6. Une comparaison plus détaillée des deux implémentations du format CSR sera présentée à la section 4.3.2. Le modèle estime le nombre de transactions avec une erreur moyenne de 2.6 %. Trois exceptions existent pour les matrices 12, 13 et 22 où la différence dépasse les 13 %. Les différences entre les résultats du modèle et du profileur sont présentées à la Figure 4.7.

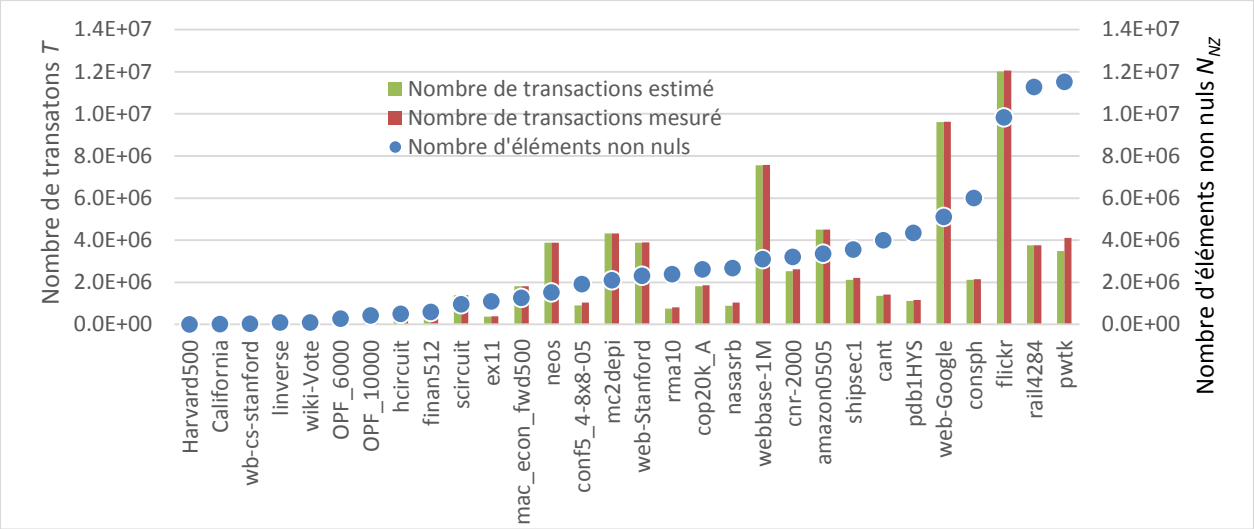


Figure 4.6 : Nombres de transactions estimés et mesurés pour le format CSR avec une chaîne par ligne comparés au nombre d'éléments non nuls de chaque matrice

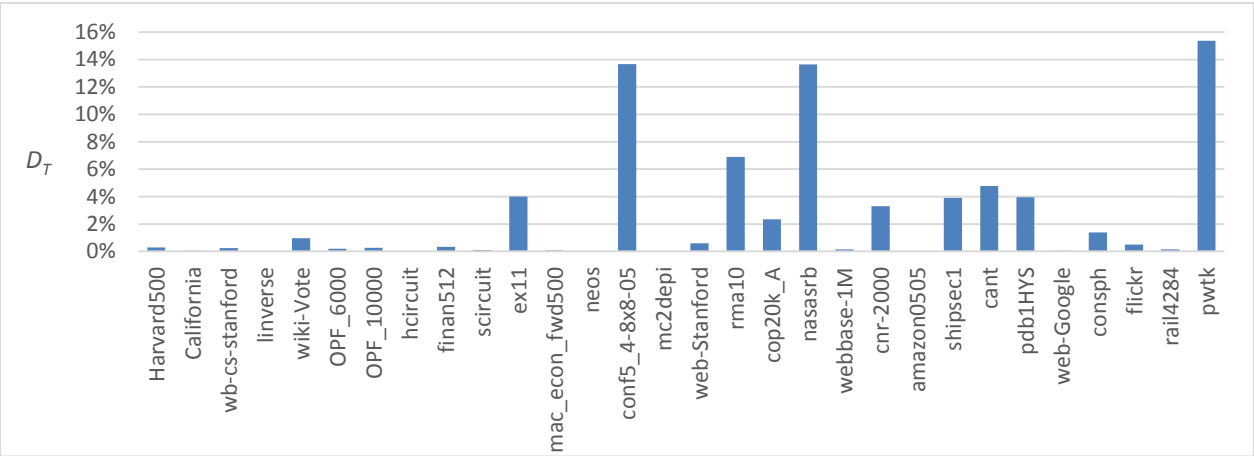


Figure 4.7 : Différences relatives entre le nombre de transactions estimé et mesuré pour le format CSR avec une chaîne par ligne de la matrice

Le modèle du format CSR avec une chaîne par ligne sépare le problème de calcul du nombre de transactions de lectures en quatre variables indépendantes : T_{ptr}, T_v, T_c et T_x. Il est possible de mesurer ces nombres séparément avec le profileur en modifiant le noyau et en mettant en commentaire certaines lectures à la mémoire. Le nombre de transactions T_{ptr} causé par la lecture au vecteur de pointeurs de ligne peut être mesuré en mettant toutes les autres lectures en commentaires. Par contre, T_v et T_c causés par la lecture au vecteur de valeur et de colonne dépendent de la lecture du pointeur de ligne et T_x causé par la lecture du vecteur x dépend de la lecture du pointeur de ligne et

de la lecture du vecteur de colonne. Avec la modification du noyau, il est possible de mesurer plusieurs nombres de lectures à la mémoire : T_{ptr}, T_{ptrv}, T_{ptrc} et T_{ptrcv}.

Il existe deux façons de mesurer les variables T_v et T_c à l'aide du profileur. Par exemple, la première façon de mesurer T_v, que nous appelerons T_{v1}, serait de mesurer la somme des lectures à la mémoire causée par le vecteur de pointeur de ligne et le vecteur de valeurs ensemble, appelé T_{ptrv} et en soustraire T_{ptr} mesuré au préalable. Nous pouvons voir comment mesurer T_{v1} et T_{c1} aux équations (4.2) et (4.3).

$$T_{v1} = T_{ptrv} - T_{ptr} \tag{4.2}$$

$$T_{c1} = T_{ptrc} - T_{ptr} \tag{4.3}$$

Selon le modèle, T_v et T_c sont égaux. Selon les résultats du profilage, T_{v1} et T_{c1} sont aussi égaux et la différence entre les valeurs estimées et mesurées est assez petite. Ces différences sont montrées à la Figure 4.8.

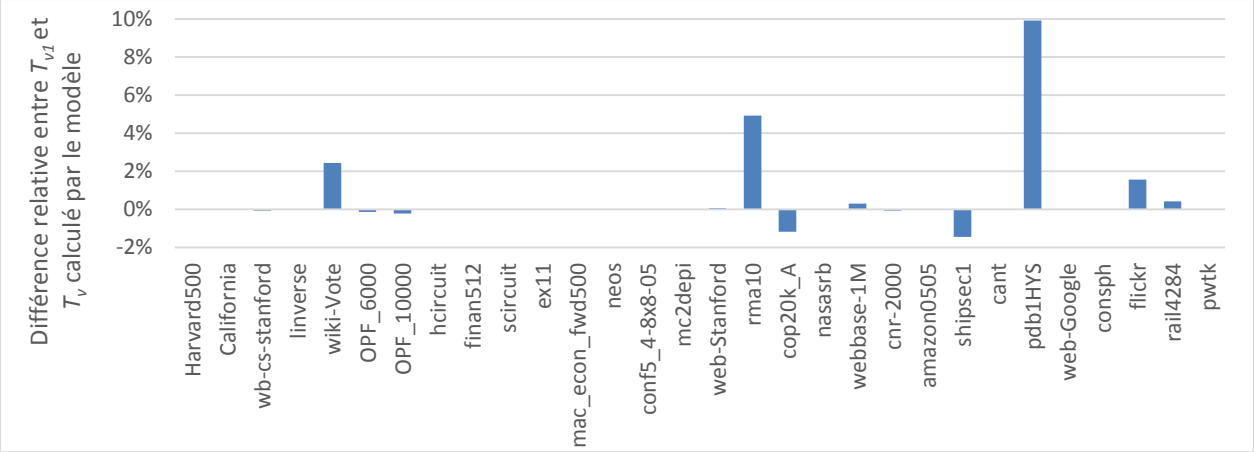


Figure 4.8 : Différences relatives entre T_{v1} mesuré par le profileur et T_v calculé par le modèle pour le format CSR avec une chaîne par ligne de la matrice

Puisque nous savons que T_v et T_c sont égaux, il est aussi possible de mesurer T_v et T_c en utilisant la variable T_{ptrvc}, comme à l'équation (4.4). Dans ce cas, T_{v1} et T_{v2} n'ont pas la même valeur et le nombre de transactions est plus élevé pour T_{v2}. La différence entre T_{v1} et T_{v2} est présentée à la Figure 4.9. Cette différence expliquerait les erreurs élevées décelées dans le nombre total de transactions à la Figure 4.7. Il semble que le profileur mesure un nombre de transactions différent

lorsque les lectures du vecteur de valeur et de colonne sont demandées dans un même noyau, ce qui n'est pas le cas de l'équation utilisée dans le modèle.

$$T_{v2} = \frac{T_{ptrcv} - T_{ptr}}{2} \tag{4.4}$$

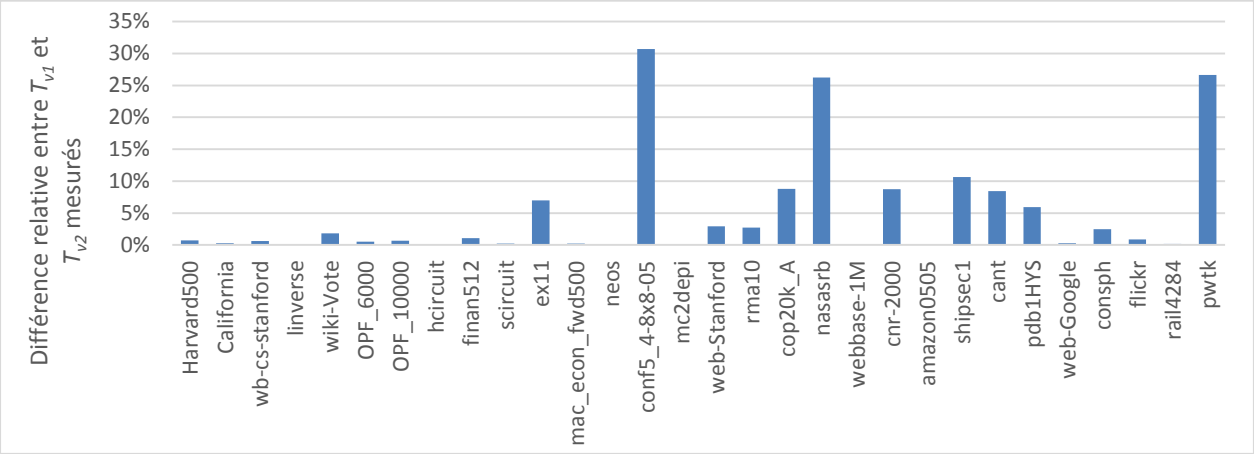


Figure 4.9 : Différences relatives entre T_{v1} et T_{v2} pour le format CSR avec une chaîne par ligne de la matrice

Les résultats du nombre de requêtes sont présentés à la Figure 4.10. En moyenne, la différence entre les résultats estimés et mesurés est de 5.16 %. On remarque des différences plus élevées que 10 % pour cinq matrices. Ces différences sont créées par le même phénomène expliqué plus haut.

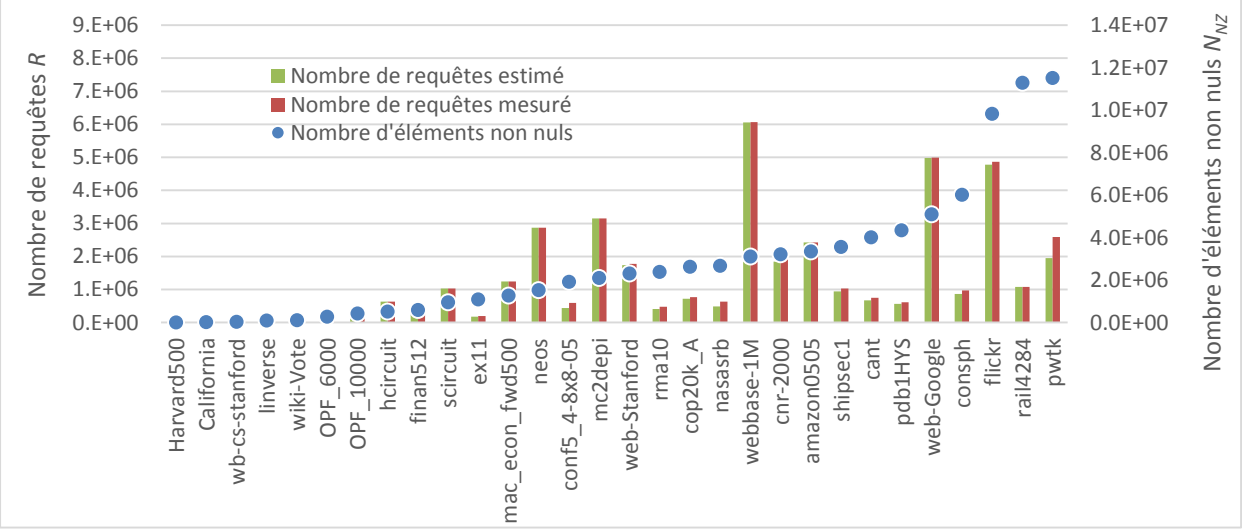


Figure 4.10 : Nombres de requêtes estimés et mesurés pour le format CSR avec une chaîne par ligne comparés au nombre d'éléments non nuls de chaque matrice

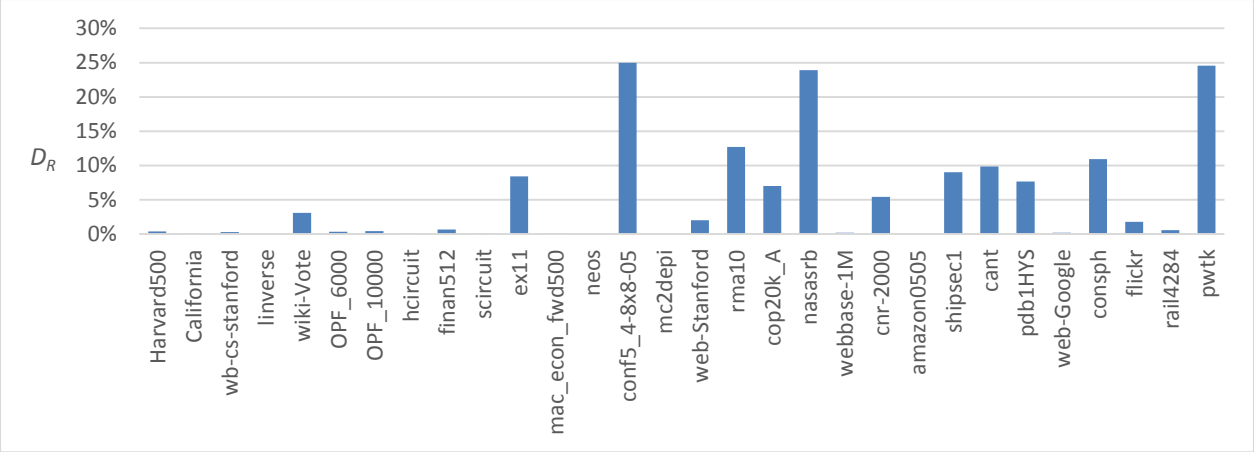


Figure 4.11 : Différences relatives entre le nombre de requêtes estimé et mesuré pour le format CSR avec une chaîne par ligne de la matrice

4.2.2 Format ELLPACK

Le format ELLPACK n'est pas un format qui compresse énormément la taille de la matrice si celle-ci ne contient que quelques lignes avec un nombre élevé d'éléments non nuls, comme expliqué à la section 2.2.4. L'exécution d'une SpMV utilisant ce format a été impossible pour certaines matrices à cause de la largeur des éléments qu'il fallait transférer à la mémoire globale du GPU. Le Tableau 4.3 indique si la taille peut tenir sur l'un ou l'autre des GPU à notre disposition en présentant le nombre de lignes *M* et le nombre maximal d'éléments sur une ligne *maxRow* de chaque matrice problématique. La taille du format est représentée par la somme des tailles des matrices *data* et *indices*. En plus du format, il faut aussi transférer à la mémoire du GPU les vecteurs *x* et *y*. Rappelons-nous que la taille de la mémoire globale des GPU utilisés est de 2048 et 4800 mégaoctets.

Tableau 4.3 : Matrices ayant un format ELLPACK de taille élevée

	Matrices	*M*	*maxRow*	Taille du format ELLPACK (Mo)	Transférable sur GTX670?	Transférable sur K20?
24	cnr-2000	325557	2716	6 746	Non	Non
28	flickr	820878	10272	64 332	Non	Non
29	web-Google	916428	456	3 188	Non	Oui
30	webbase-1M	1000005	4700	35 858	Non	Non

Tous les résultats de cette section sont présentés en détail à l'Annexe C. Pour le format ELL, les matrices ont été mises en ordre croissant de la taille d'une des matrices du format. Un graphique représentant le nombre de transactions estimé et mesuré des matrices est présenté à la Figure 4.12.

Les nombres de transactions estimés par le modèle du format ELLPACK sont égaux aux nombres de transactions mesurés. Deux exceptions sont les matrices *rail4284* et *mc2depi* dont les différences se trouvent à la Figure 4.13. Le nombre de transactions n'est pas proportionnel au nombre d'éléments non nuls comme il l'était pour le format CSR avec un fil par ligne. On remarque plutôt une relation avec la taille des matrices *data* et *indices* qui ont une taille de *M* par *K*. Les matrices sont donc dans cet ordre dans les figures.

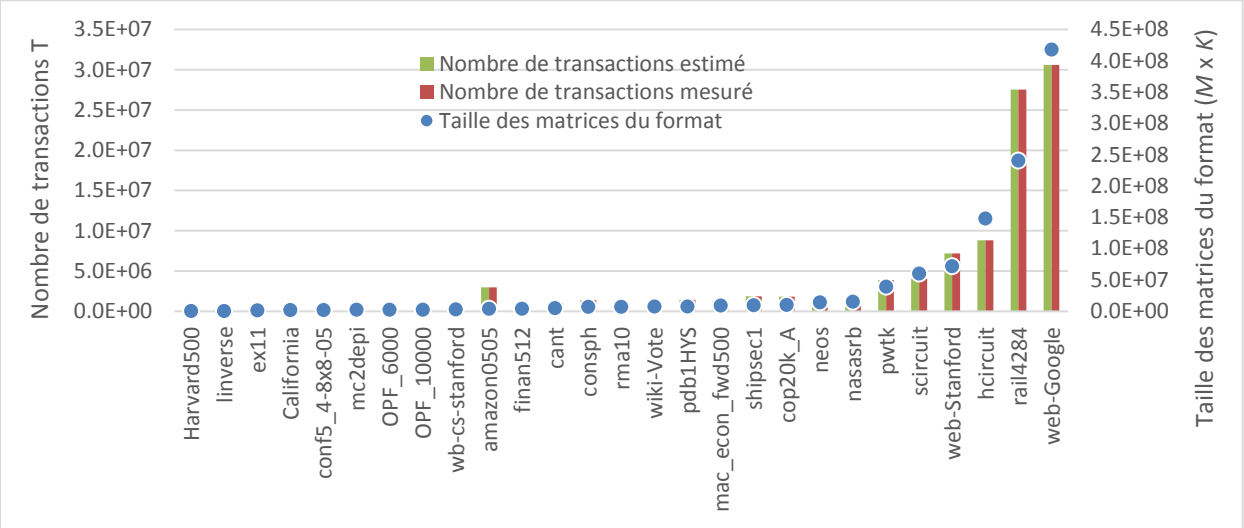


Figure 4.12 : Nombres de transactions estimés et mesurés pour le format ELLPACK comparés à la taille des matrices *data* et *indices* de chaque matrice

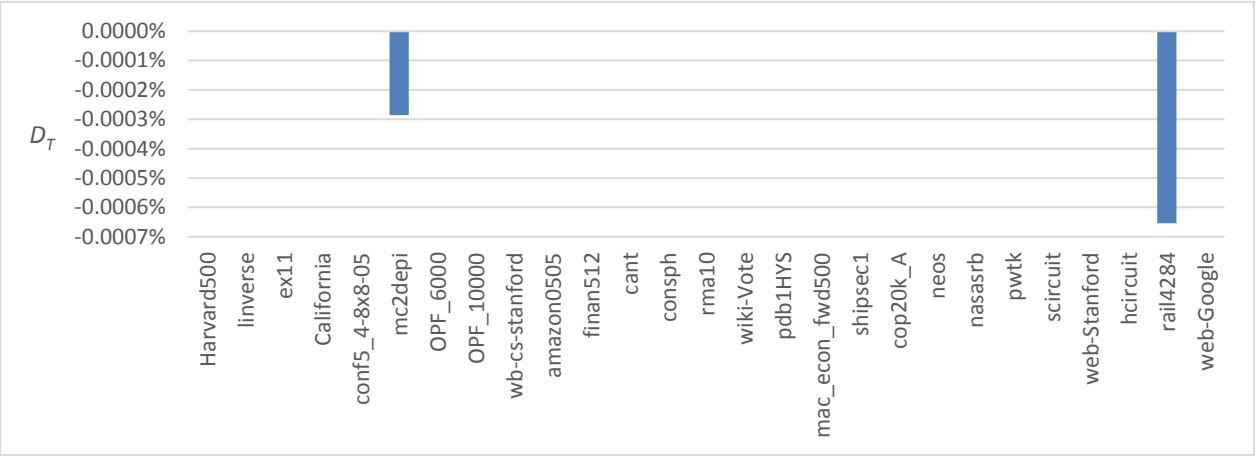


Figure 4.13 : Différences relatives entre le nombre de transactions estimé et mesuré pour le format ELL

Le nombre de requêtes estimé, quant à lui, présente une grande différence par rapport à ce qui est mesuré par le profileur. Les résultats des nombres de requêtes estimé et mesuré sont présentés à la

Figure 4.14. La différence se trouve dans le calcul de R_i et R_x. Une modification du noyau permet de recueillir le nombre de requêtes décomposé. Celui-ci est présenté à la Figure 4.16.

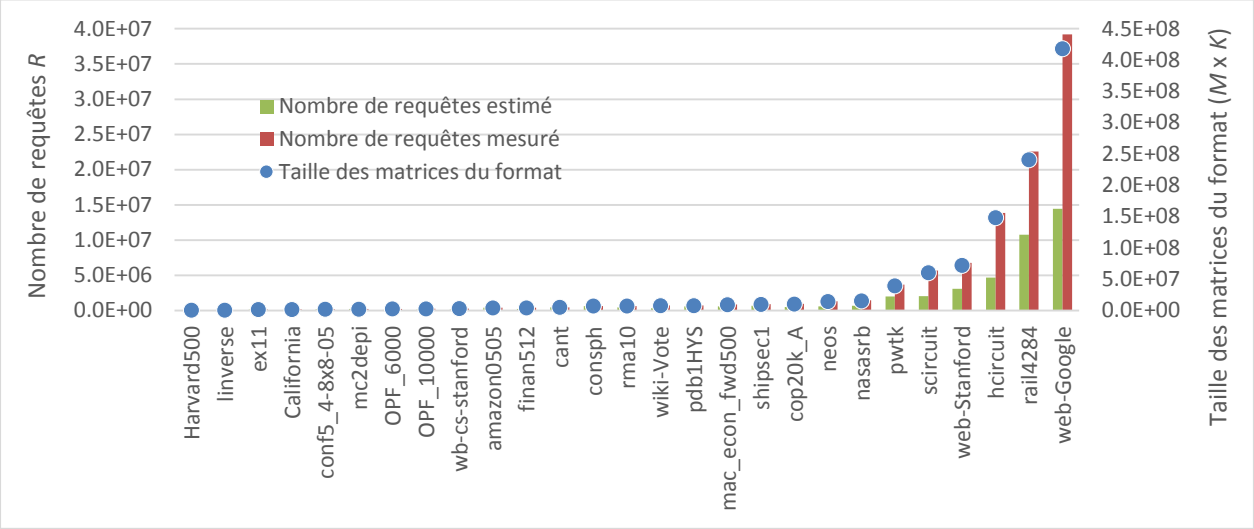


Figure 4.14 : Nombres de requêtes estimés et mesurés pour le format ELLPACK comparés à la taille des matrices *data* et *indices* de chaque matrice

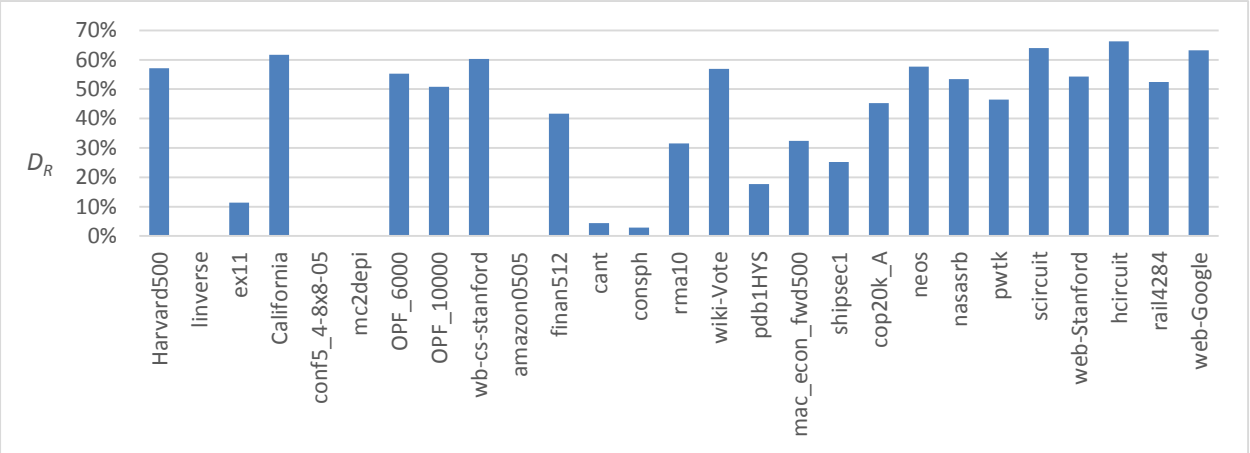


Figure 4.15 : Différences relatives entre le nombre de requêtes estimé et mesuré pour le format ELL

Afin de mesurer les nombres R_d, R_i, R_x et R_y, il faut exécuter une partie de l'application. Par exemple, pour avoir le nombre de requêtes R_d, il suffit de mettre en commentaires les lectures à la matrice *indices* et au vecteur x. Par contre, il faut s'assurer que l'écriture au vecteur y utilise les lectures réalisées à la matrice *data*. Sinon, le compilateur éliminera les accès à la mémoire inutiles pour optimiser le noyau.

On remarque que les requêtes à la matrice *data* estimées et mesurées sont identiques. Par contre, les requêtes à la matrice *indices* et au vecteur *x* sont différentes. Les requêtes mesurées par le profileur pour la lecture de ces deux dernières régions de la mémoire sont les mêmes que R_d. Le profileur semble donc croire qu'il y a autant d'exécution de l'instruction de lecture à la matrice *data* qu'à la matrice *indices* et au vecteur *x*. Rappelons-nous que le code avait été écrit de sorte à minimiser les lectures à ses deux dernières régions de la mémoire en vérifiant d'abord si la valeur lue dans la matrice *data* est non nulle (Figure 3.8). La documentation du profileur [22] ne donne aucune indication sur les métriques « gld request » et « gst request » qui sont utilisées ici pour les mesures. Une possible hypothèse est que le profileur estime la valeur du nombre de requêtes et qu'il considère que la condition à laquelle la valeur lue dans *data* est non nulle sera toujours vraie pour au moins un fil d'exécution par chaîne. En effet, le profileur estime aussi le nombre d'exécutions des instructions en assembleur où sont lues les éléments de la matrice *indices* et du vecteur *x*. Les nombres d'exécutions sont les mêmes que le nombre d'exécutions estimé de l'instruction de lecture de la matrice *data*. Il se peut aussi que l'estimation du profileur soit bonne et que l'instruction de lecture des éléments de la matrice *indices* et du vecteur *x* soit effectuée avant même que la condition soit vérifiée. Par contre, le nombre de transactions a été bien estimé par le modèle proposé, qui ne considère pas que les instructions de transfert des données. Il semble donc y avoir une contradiction entre les nombres de transactions et de requêtes mesurés.

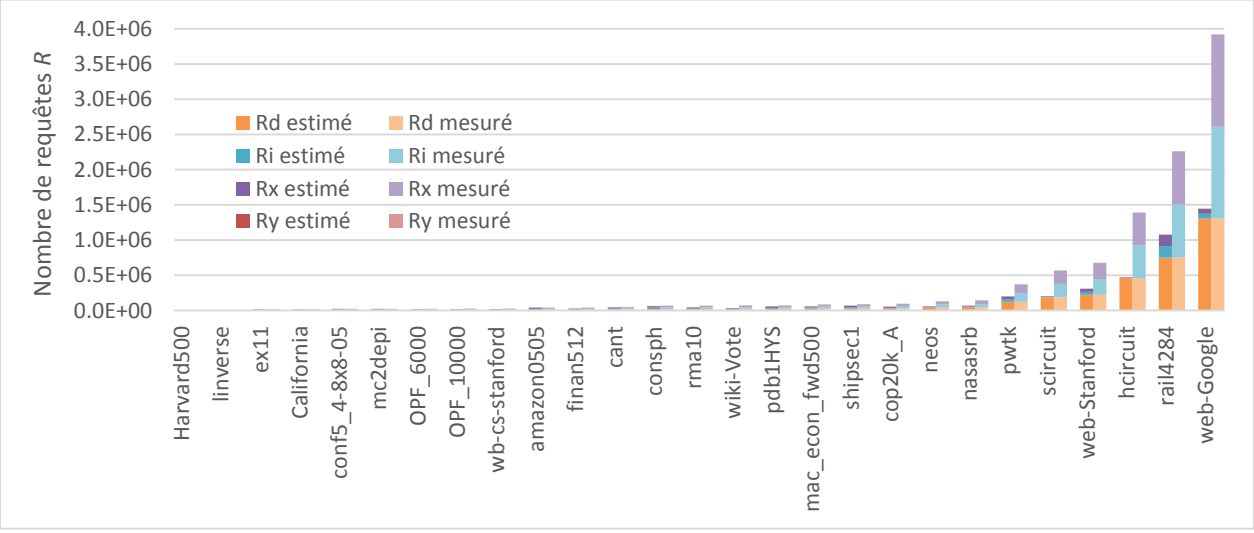


Figure 4.16 : Nombres de requêtes décomposés estimés et mesurés pour le format ELLPACK

Pour ce qui est des écritures au vecteur *y*, le nombre de requêtes estimé et mesuré est le même. Les nombres exacts qui ont permis de faire ce graphique se trouvent à l'Annexe C.

4.2.3 Format COO

Le format COO requiert l'appel à trois noyaux. Les résultats de chacun d'entre eux seront présentés séparément aux sections suivantes. De plus, tous les résultats exacts sont présentés à l'Annexe D. Ici, nous présenterons les résultats totaux des transactions et requêtes nécessaires par une SpMV ayant une matrice stockée avec le format COO.

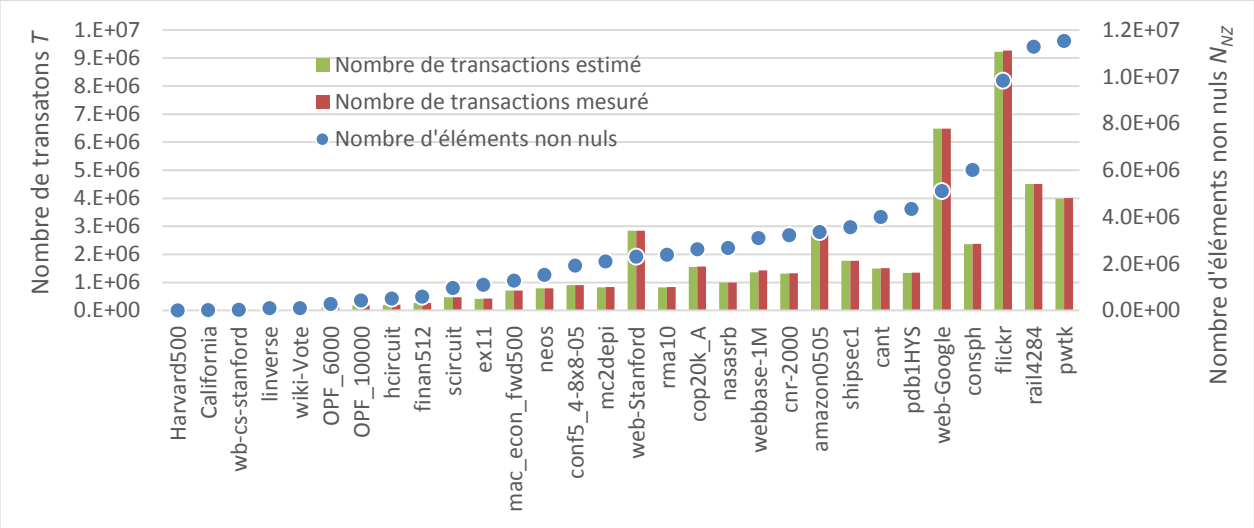


Figure 4.17 : Nombres de transactions estimés et mesurés pour le format COO comparés au nombre d'éléments non nuls de chaque matrice

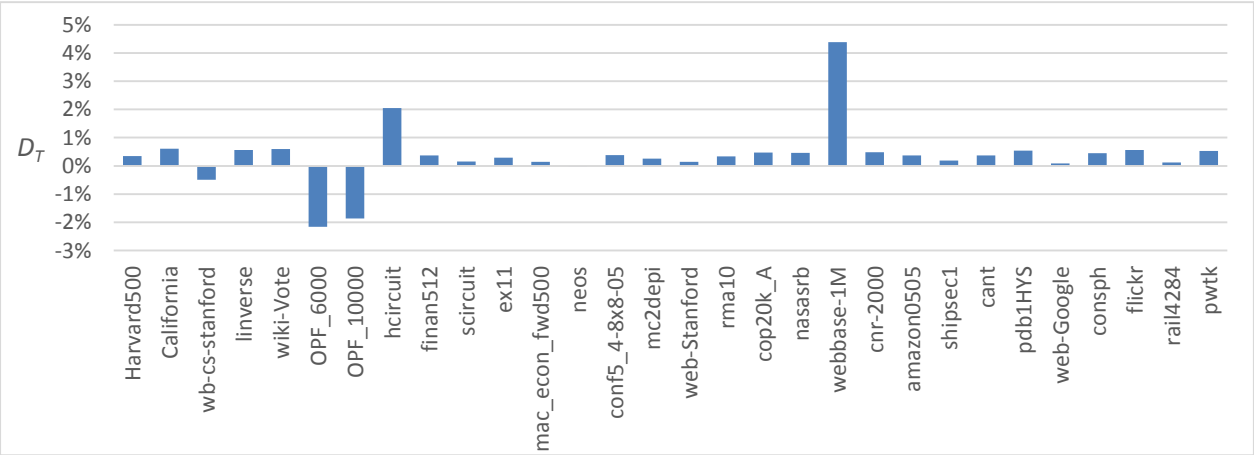


Figure 4.18 : Différences relatives entre le nombre de transactions estimé et mesuré pour le format COO

Le nombre de transactions du format COO est présenté à la Figure 4.17 avec les matrices placées en ordre de leur nombre d'éléments non nuls. Les différences entre les nombres estimés et mesurés sont présentées à la Figure 4.18. La moyenne des erreurs absolues est de 0.64 % avec une erreur maximale de 4.38 %. Pour le nombre de requêtes, les résultats sont présentés à la Figure 4.19 et les différences à la Figure 4.20. La moyenne des erreurs est de 0.68 %. On remarque que le modèle surestime le nombre de requêtes pour toutes les matrices testées. Les erreurs par rapport au nombre de transactions et de requêtes sont dues essentiellement à l'estimation du deuxième noyau qui s'occupe d'additionner les résultats temporaires. Nous en discuterons à la section 4.2.3.2.

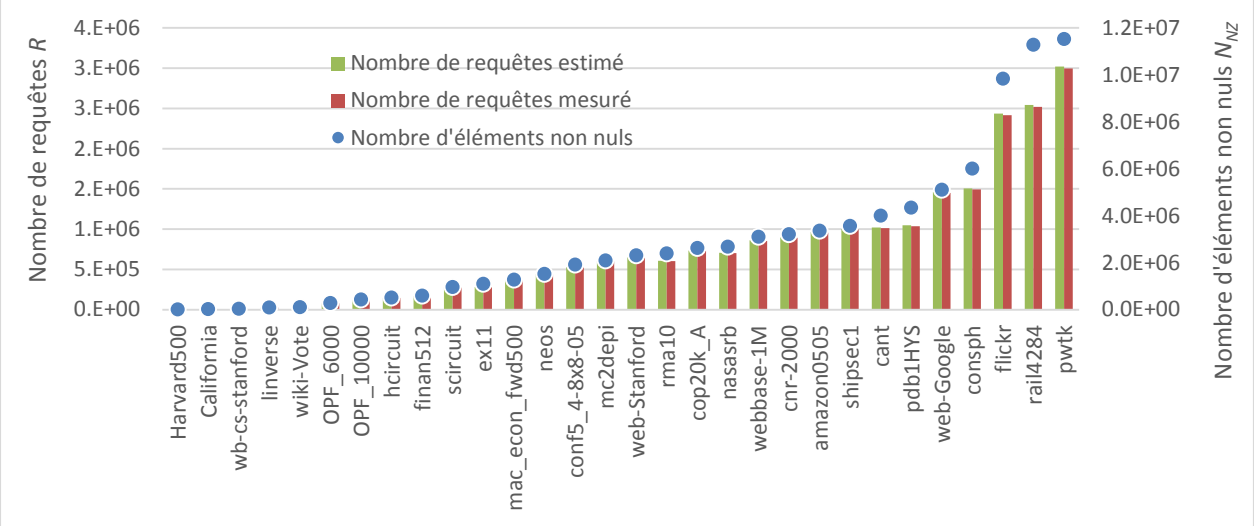


Figure 4.19 : Nombres de requêtes estimés et mesurés pour le format COO comparés au nombre d'éléments non nuls de chaque matrice

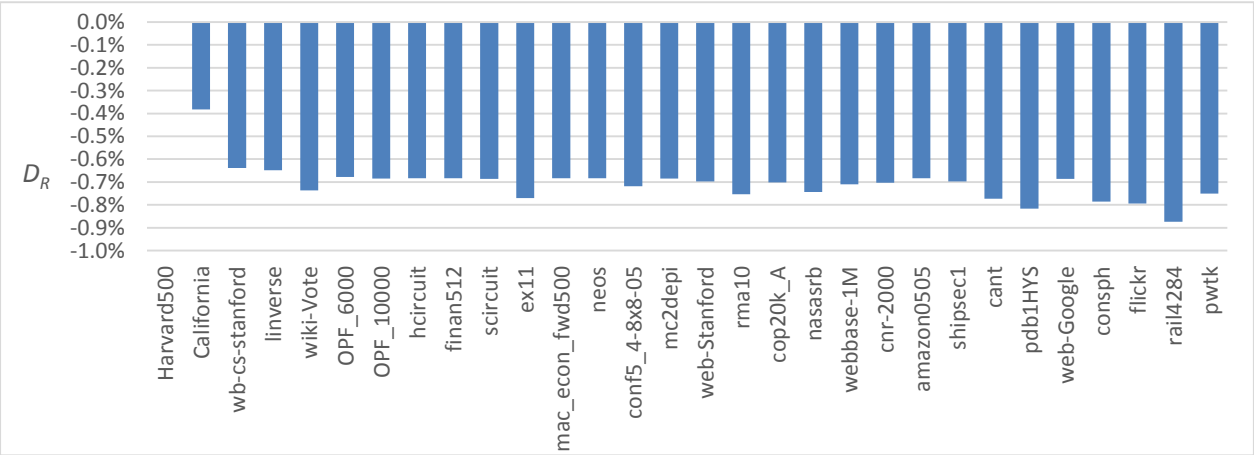


Figure 4.20 : Différences relatives entre le nombre de requêtes estimé et mesuré pour le format COO

4.2.3.1 SpMV parallèle

Les transactions à la mémoire du premier noyau composant l'algorithme de SpMV de format COO ont été modélisées presque parfaitement. Quelques erreurs d'une ou deux transactions sont produites pour cinq matrices représentées à la Figure 4.21. Le nombre de requêtes est tout aussi exact excepté pour quatre matrices pour lesquelles le nombre de requêtes estimé diffère de deux requêtes en trop. Les différences sont présentées à la Figure 4.22.

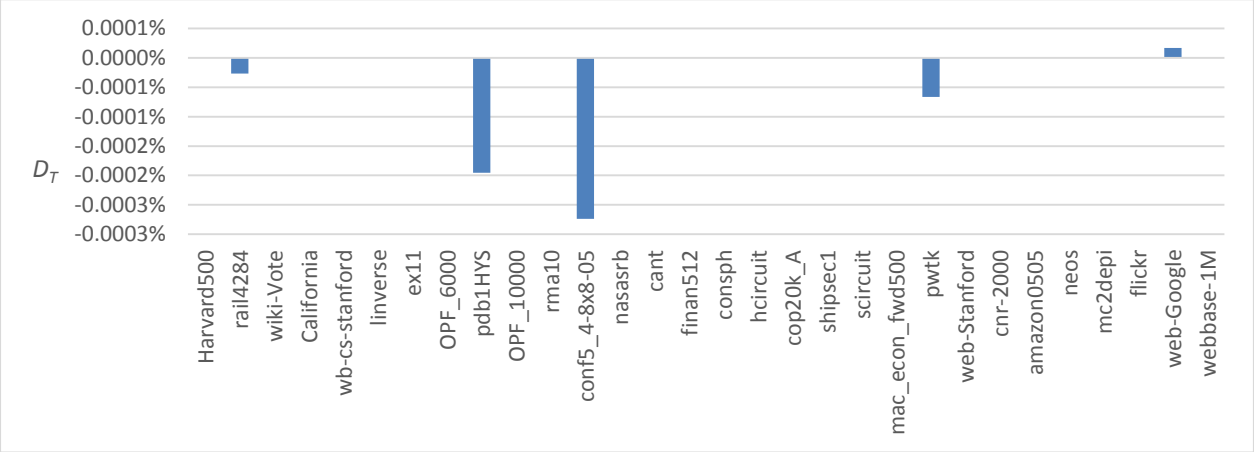


Figure 4.21 : Différences relatives entre le nombre de transactions estimé et mesuré pour le noyau de SpMV parallèle du format COO

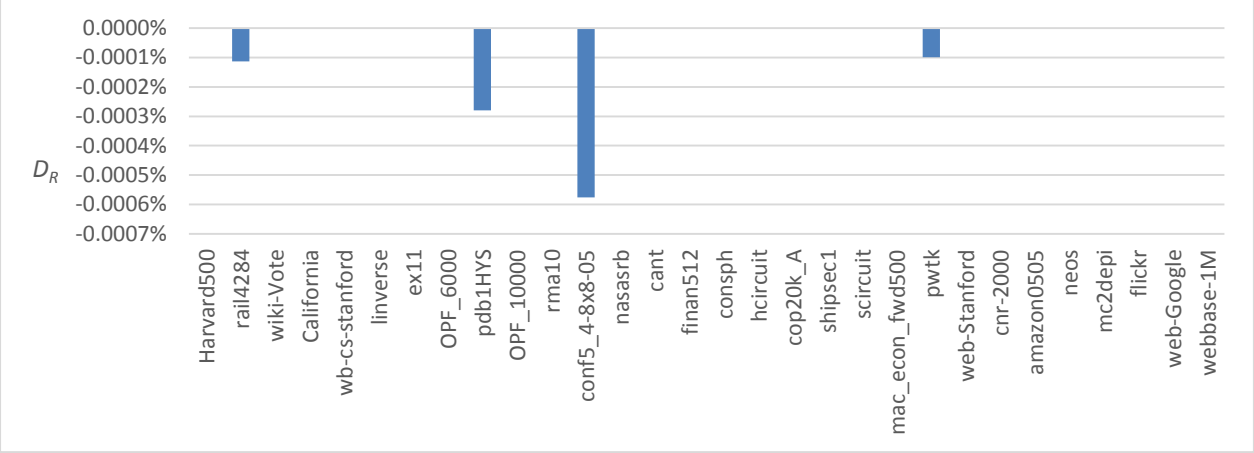


Figure 4.22 : Différences relatives entre le nombre de requêtes estimé et mesuré pour le noyau de SpMV parallèle du format COO

4.2.3.2 Addition des résultats temporaires

Le noyau exécutant l'addition des résultats temporaires est celui qui cause le plus d'erreurs pour le format COO au complet. Comme nous l'avons vu à la section 3.4.2, il était nécessaire de faire une estimation des valeurs contenues dans les vecteurs temporaires mesurés au noyau précédent. Il est difficile de connaître le contenu de ces vecteurs sans exécuter le programme de façon dynamique. Par contre, ce noyau est exécuté sur un seul bloc et réalise peu de transactions et de requêtes par rapport au noyau précédent. Les erreurs recueillies dans cette étape sont donc négligeables.

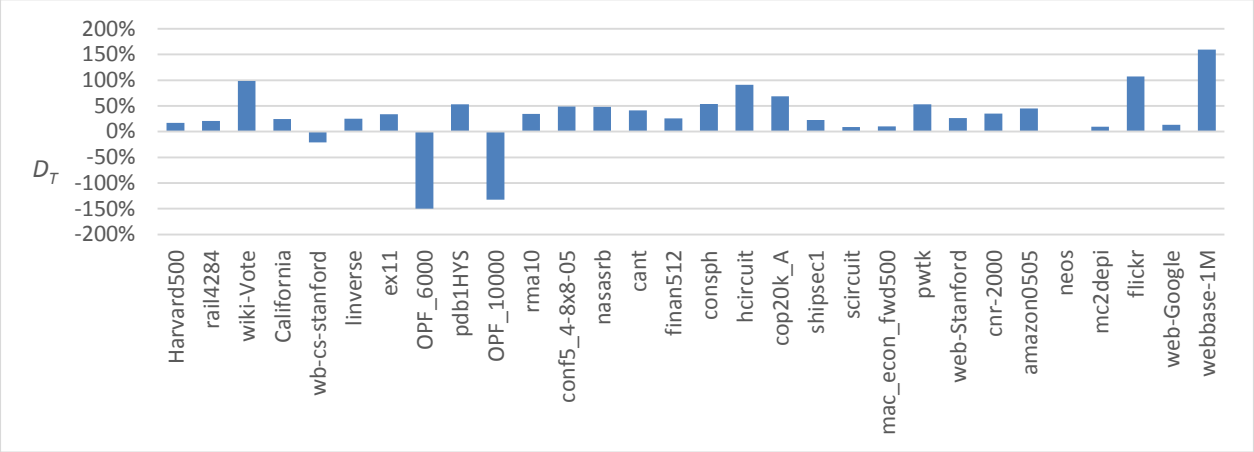


Figure 4.23 : Différences relatives entre le nombre de transactions estimé et mesuré pour le noyau d'addition des résultats temporaires du format COO

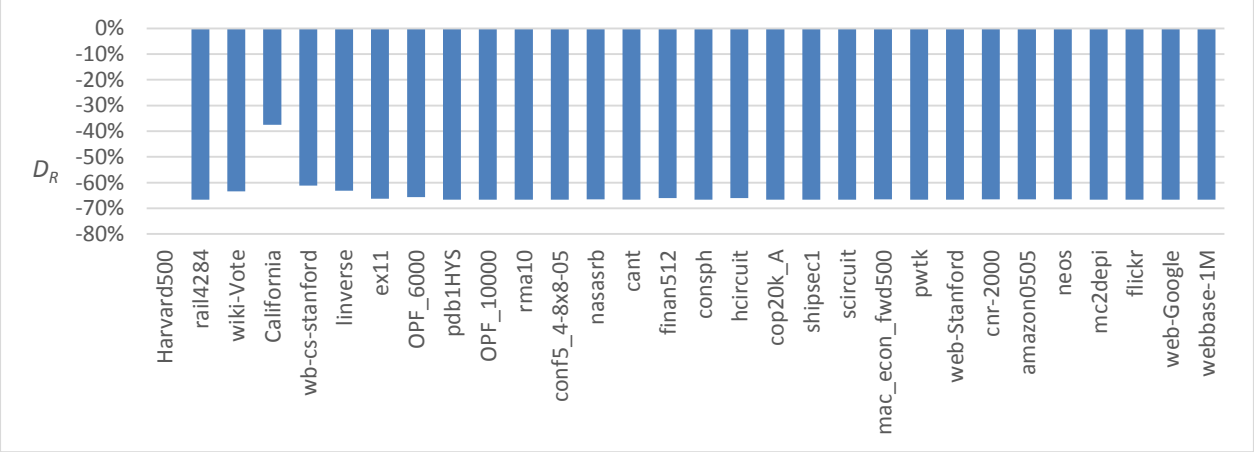


Figure 4.24 : Différences relatives entre le nombre de requêtes estimé et mesuré pour le noyau d'addition des résultats temporaires du format COO

4.2.3.3 SpMV séquentielle

Le dernier noyau de l'implémentation de la SpMV utilisant le format COO effectue une multiplication séquentielle des derniers éléments mis de côté avec le vecteur. Ce noyau est très facile à estimer, il n'est donc pas étonnant que les nombres de transactions et de requêtes estimés concordent exactement à ceux mesurés.

4.2.4 Format hybride ELL/COO

Le format HYB permet d'utiliser le format ELL de façon plus optimale en permettant de ne pas être affecté par les lignes qui auraient beaucoup plus d'éléments non nuls que le reste de la matrice. Ainsi, un nouveau *K* est calculé, appelé *K* optimal (*Kopt*), qui fait en sorte qu'au moins un tiers des lignes sont de cette longueur (voir section 2.2.5). Si la valeur optimale de *K* est identique au nombre d'éléments non nuls de la plus longue ligne, il n'y a aucun élément non nul de la matrice qui est stocké en format COO. Ceci est le cas de cinq matrices : *linverse*, *conf5_4-8x8-05*, *consph*, *amazon0505* et *mc2depi*. Ces matrices sont traitées de la même façon que pour le format ELL et auront des résultats identiques à ceux de la section 4.2.2. Les valeurs de *K* optimales utilisées dans cette section sont présentées à la Figure 4.25 en comparaison avec les valeurs de *K* du format ELL et le nombre de colonnes de la matrice creuse originale.

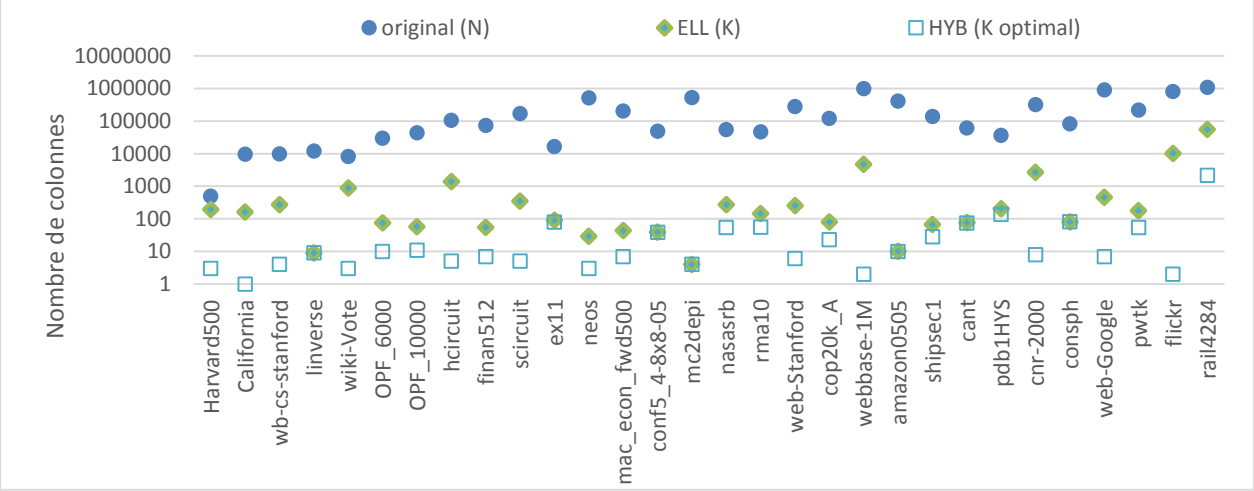


Figure 4.25 : Nombres de colonnes des formats HYB et ELL et de la matrice originale

Tous les résultats de cette section sont présentés à l'Annexe E. Les nombres de transactions sont présentés à la Figure 4.26 dans l'ordre de la taille des matrices en format HYB. La taille du format est mesurée en additionnant la taille des matrices du format ELL par la taille des vecteurs du format

COO. Les matrices *indices* et *data* ont *M* lignes et *Kopt* colonnes et les trois vecteurs du format COO ont chacun autant d'éléments que le nombre d'éléments non nuls qui ne sont pas placés dans le format ELL. Ce nombre se calcule en parcourant le nombre d'éléments non nuls par ligne de chaque matrice et en soustrayant par *K* optimal. Le nombre de transactions estimé présente une moyenne d'erreur absolue de 0.35 %. Les différences relatives sont présentées à la Figure 4.27.

Le nombre de requêtes est présenté à la Figure 4.28. Les différences relatives entre les requêtes estimées et mesurées, présentées à la Figure 4.29, ont une moyenne absolue de 1.48 %.

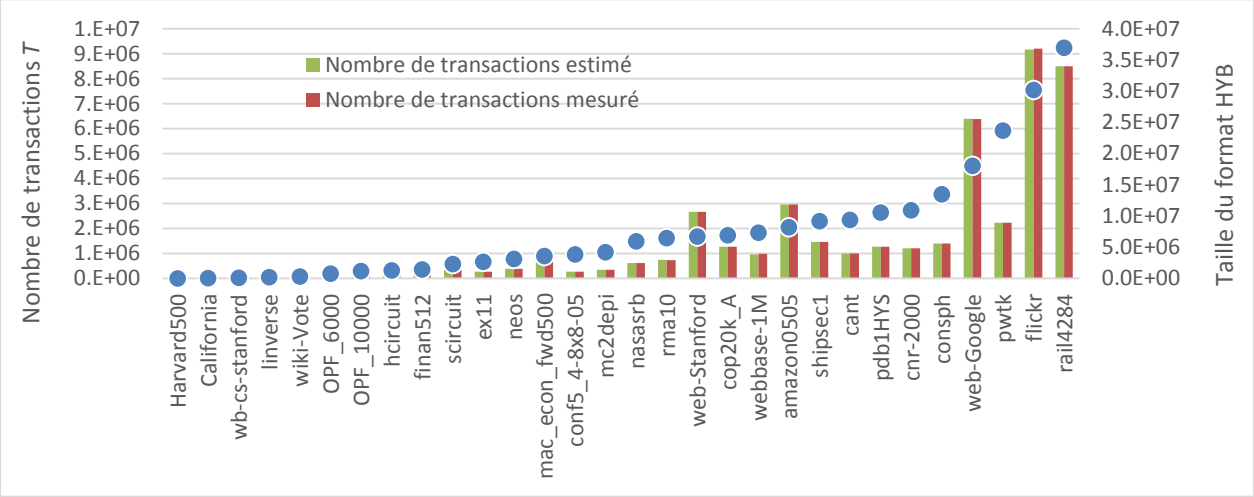


Figure 4.26 : Nombres de transactions estimés et mesurés pour le format HYB comparé à la taille de chaque matrice formatée

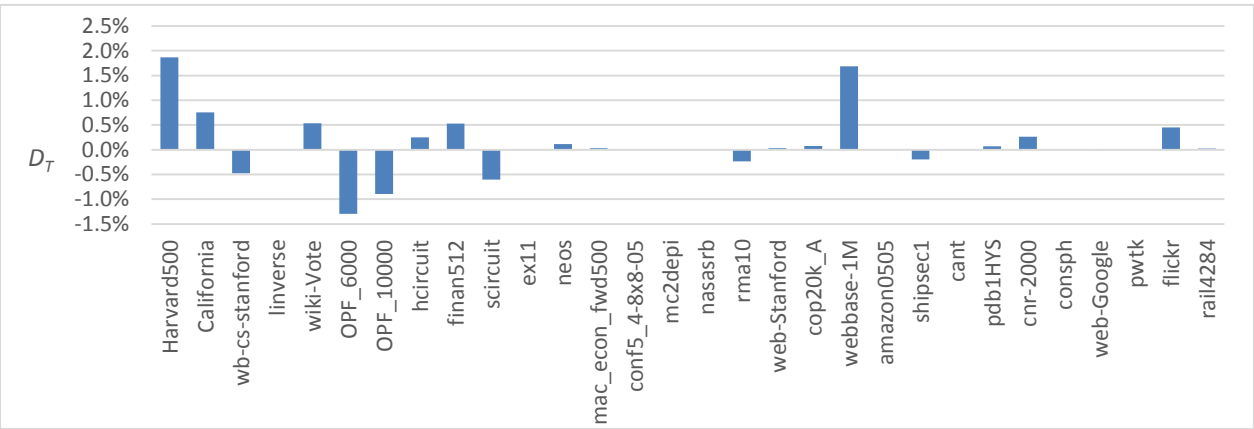


Figure 4.27 : Différences relatives entre le nombre de transactions estimé et mesuré pour le format HYB

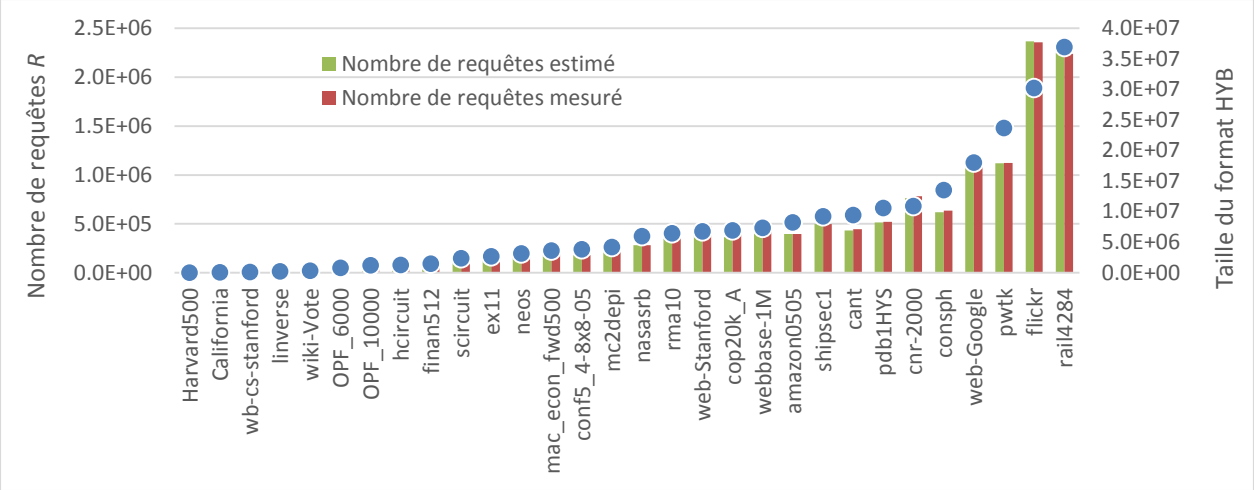


Figure 4.28 : Nombres de requêtes estimés et mesurés pour le format HYB comparé à la taille de chaque matrice formatée

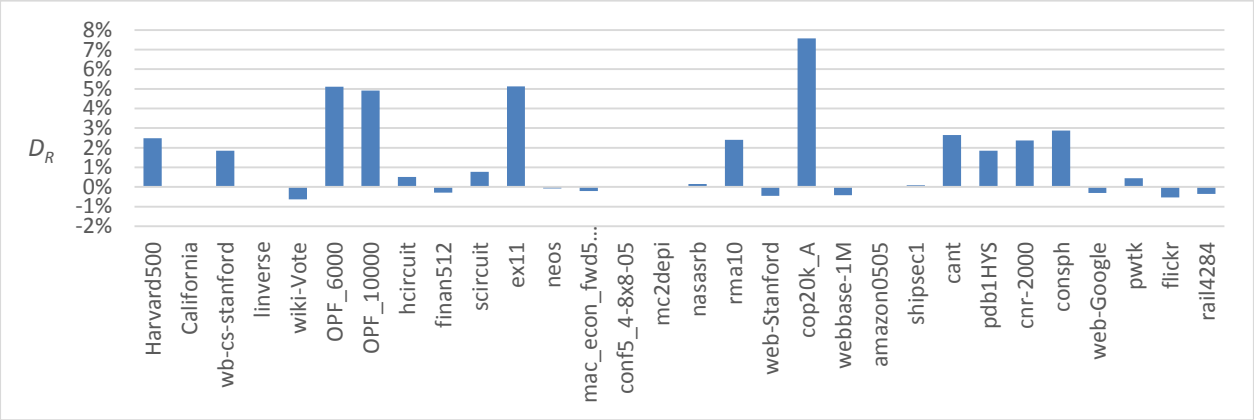


Figure 4.29 : Différences relatives entre les nombres de requêtes estimés et mesurés pour le format HYB

4.2.4.1 Portion ELLPACK du format HYB

Les nombres de transactions estimés et mesurés pour la portion ELL du format HYB sont présentés à la Figure 4.30. Les matrices sont placées en ordre de la taille d'une des matrices du format ELL, *data* ou *indices*, avec la nouvelle valeur de K. Les nombres de transactions de la section ELL sont, à une seule exception, exacts et les erreurs relatives sont présentées à la Figure 4.31. Pour les requêtes, on remarque à la Figure 4.32 que R_d mesuré et estimé sont les mêmes et que R_i et R_x diffèrent de ce qui est attendu. Par contre, les erreurs sont moins flagrantes qu'à la section 4.2.2. En effet, la nouvelle valeur de K permet d'avoir moins de valeurs nulles stockées dans la matrice

data. Ainsi, il y a moins de chance qu'une chaîne complète ait à lire des valeurs nulles, ne demandant aucun accès à la mémoire pour des indices de colonnes et un élément du vecteur *x*. Les différences relatives sont présentées à la Figure 4.33.

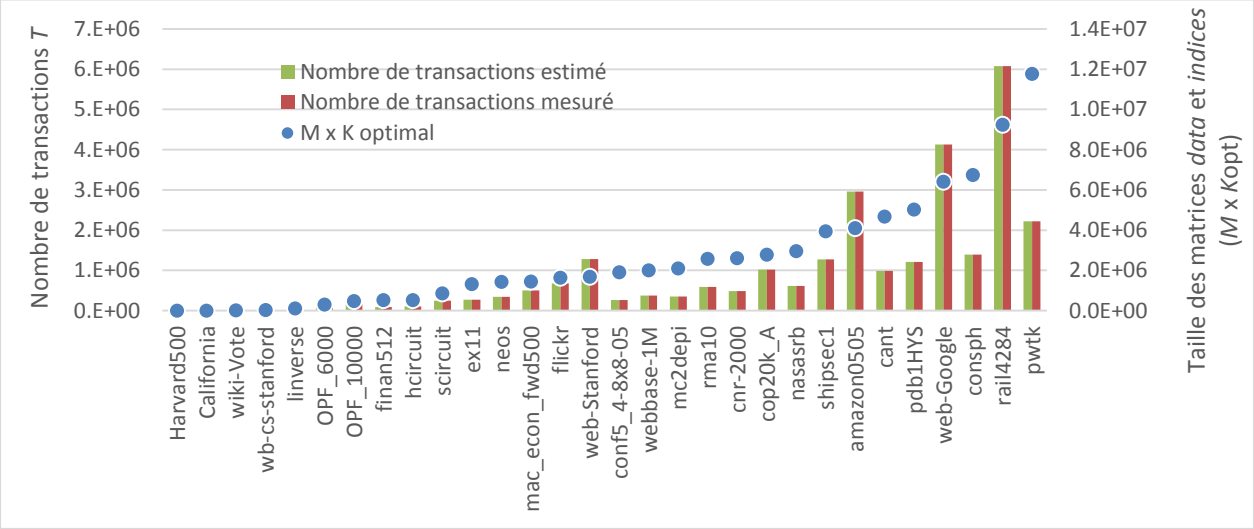


Figure 4.30 : Nombres de transactions estimés et mesurés pour la section ELL du format HYB comparé à la taille des matrices *data* et *indices*

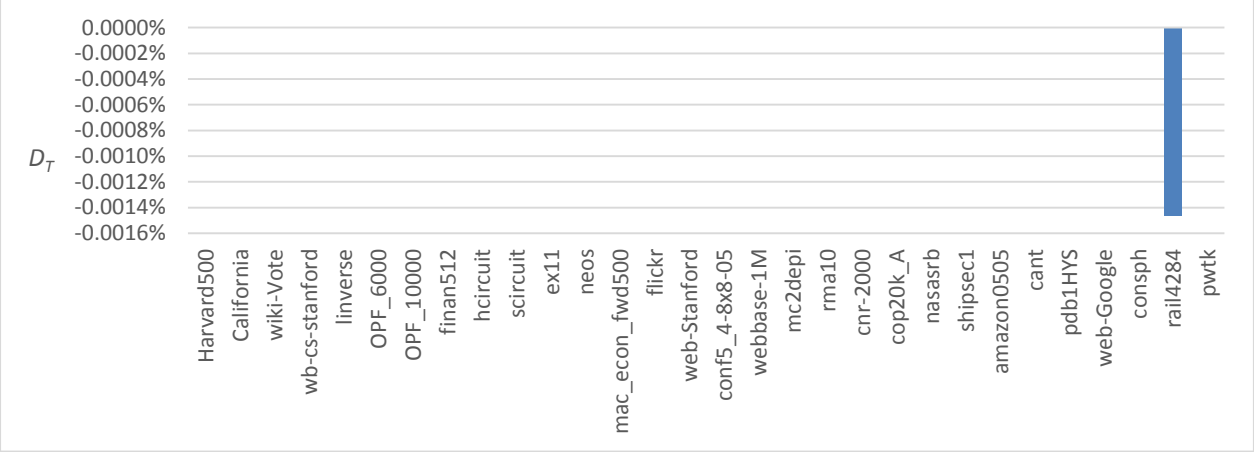


Figure 4.31 : Différences relatives entre les nombres de transactions estimés et mesurés pour la section ELL du format HYB

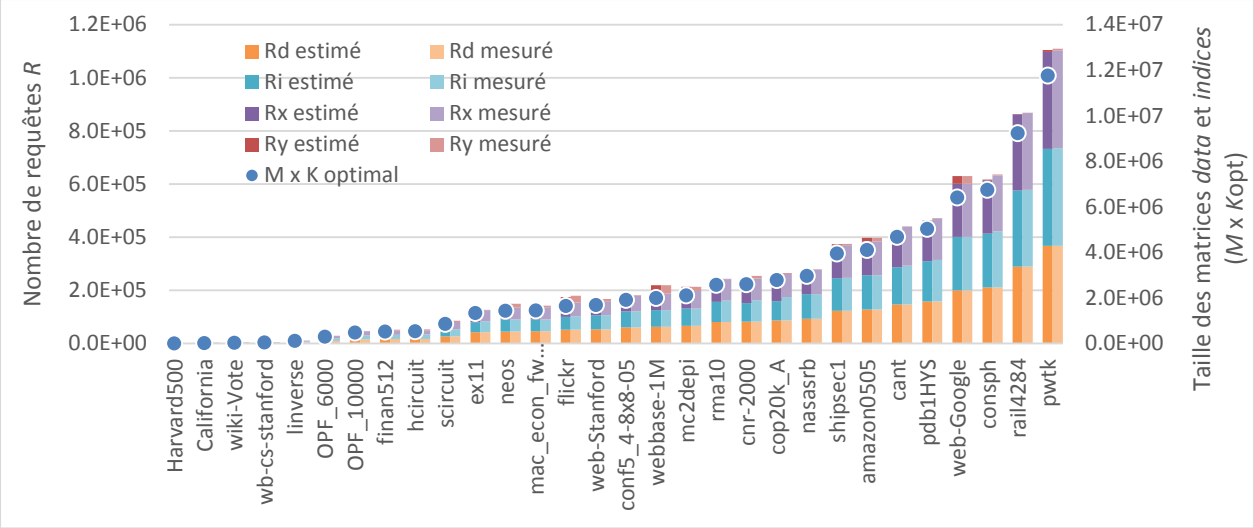


Figure 4.32 : Nombres de requêtes estimés et mesurés pour la section ELL du format HYB comparé à la taille des matrices *data* et *indices*

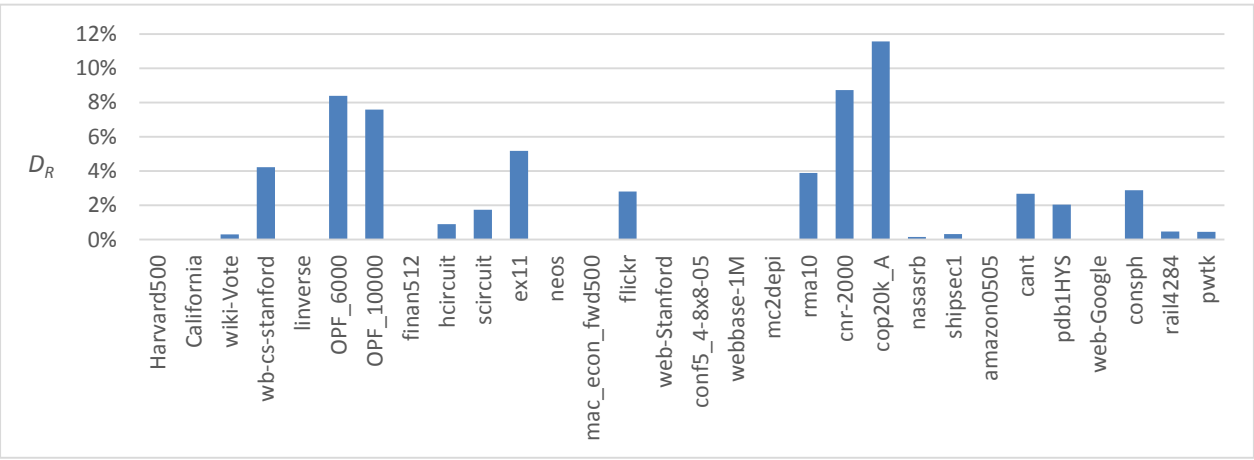


Figure 4.33 : Différences relatives entre les nombres de requêtes estimés et mesurés pour la section ELL du format HYB

4.2.4.2 Portion COO du format HYB

Le nombre de transactions de cette section est présenté à la Figure 4.34 et les différences relatives à la Figure 4.35. Le nombre de requêtes et les différences relatives sont à la Figure 4.36 et la Figure 4.37 respectivement. Les matrices de la portion en format COO ont été placées en ordre croissant du nombre d'éléments non nuls. On remarque surtout une relation entre le nombre d'éléments non nuls et le nombre de requêtes.

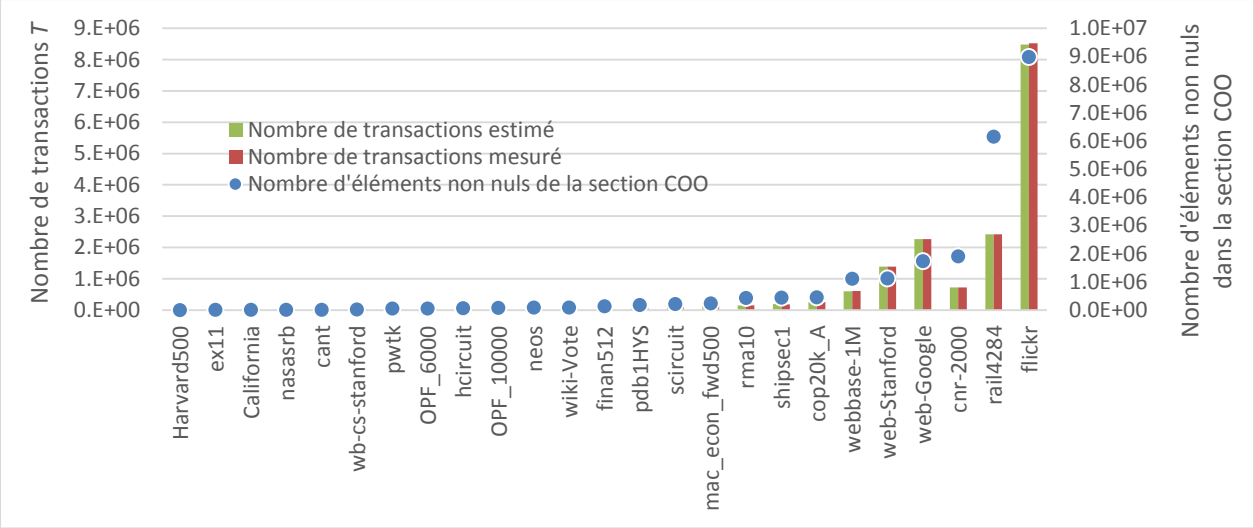


Figure 4.34 : Nombres de transactions estimés et mesurés pour la portion COO du format HYB comparé au nombre d'éléments non nuls de la section

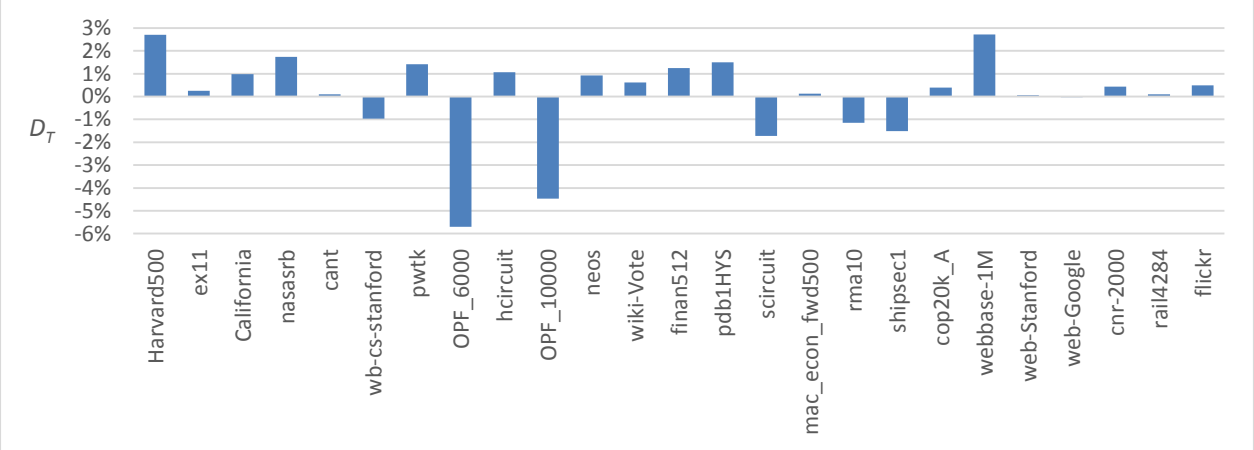


Figure 4.35 : Différences relatives entre les nombres de transactions estimés et mesurés pour la section COO du format HYB

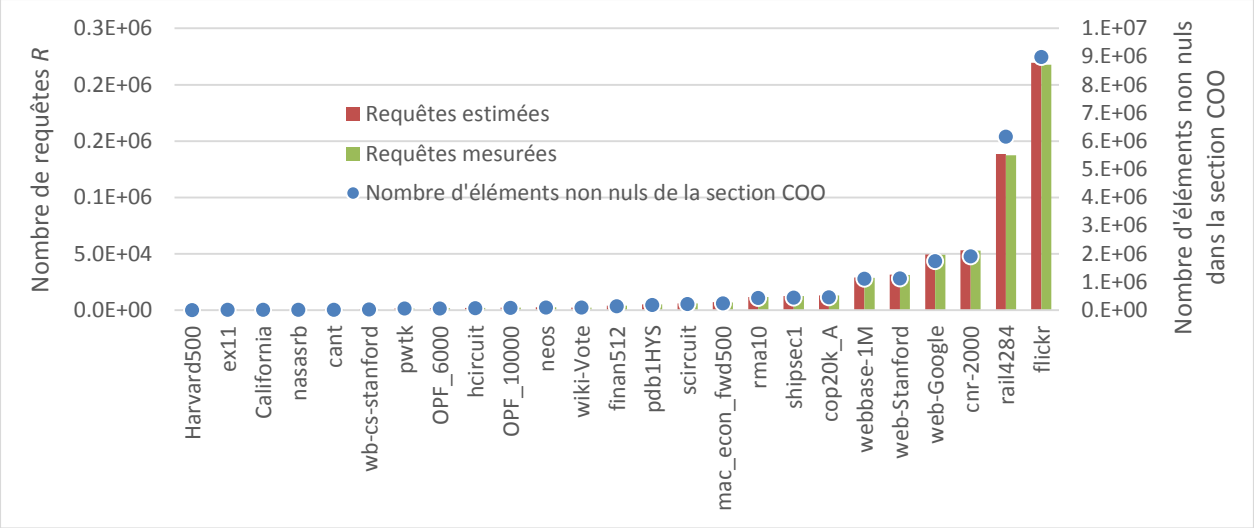


Figure 4.36 : Nombres de requêtes estimés et mesurés pour la section COO du format HYB comparé au nombre d'éléments non nuls de la section

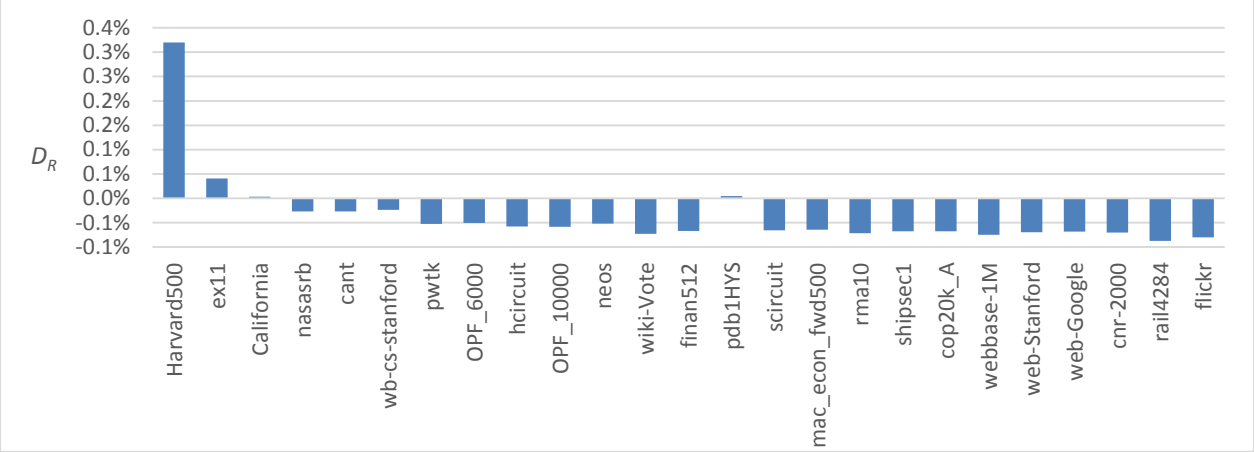


Figure 4.37 : Différence entre le nombre de requêtes estimé et mesuré pour la section COO du format HYB

4.3 Discussion

Dans cette section, nous discuterons des résultats des cinq implémentations basées sur : 1) le format CSR avec un fil par ligne de la matrice (que nous appellerons CSR-t), 2) le format CSR avec une chaîne par ligne de la matrice (que nous appellerons CSR-w), 3) le format ELL, 4) le format COO et 5) le format HYB. Nous verrons d'abord un résumé de la performance du modèle par rapport aux mesures du profileur. Puis, nous comparerons les implémentations en termes de leur nombre

de transactions et de requêtes et discuterons de la meilleure implémentation pour une matrice donnée. Enfin, nous verrons comment le temps d'exécution réagit selon les différentes implémentations et nous proposerons une équation permettant d'estimer le temps d'exécution d'une SpMV.

4.3.1 Performance du modèle

Le modèle proposé estime en général très bien le nombre de transactions et de requêtes à la mémoire. Un résumé des valeurs absolues des différences relatives entre les nombres de transactions estimés et mesurés est présenté à la Figure 4.38. Les nombres de transactions pour le format ELL sont estimés exactement pour presque toutes les matrices, c'est pourquoi on ne voit que deux points pour ce format sur la figure ayant une ordonnée à l'échelle logarithmique. Comme nous l'avons vu à la section des résultats, les nombres de transactions du format CSR avec une chaîne par ligne de la matrice (CSR-w) est celui qui donne les résultats les moins bons, avec un maximum atteignant les 15 %. L'implémentation du format CSR avec un fil par ligne (CSR-t) donne des erreurs irrégulières. Pour certaines matrices dont *pdb1HYS*, *consph* ou *ex11*, l'erreur est la plus faible et pour d'autres matrices comme *mac_econ_fwd500*, l'erreur est la plus élevée. En général, la moyenne des erreurs est autour de 1 % pour tous les formats.

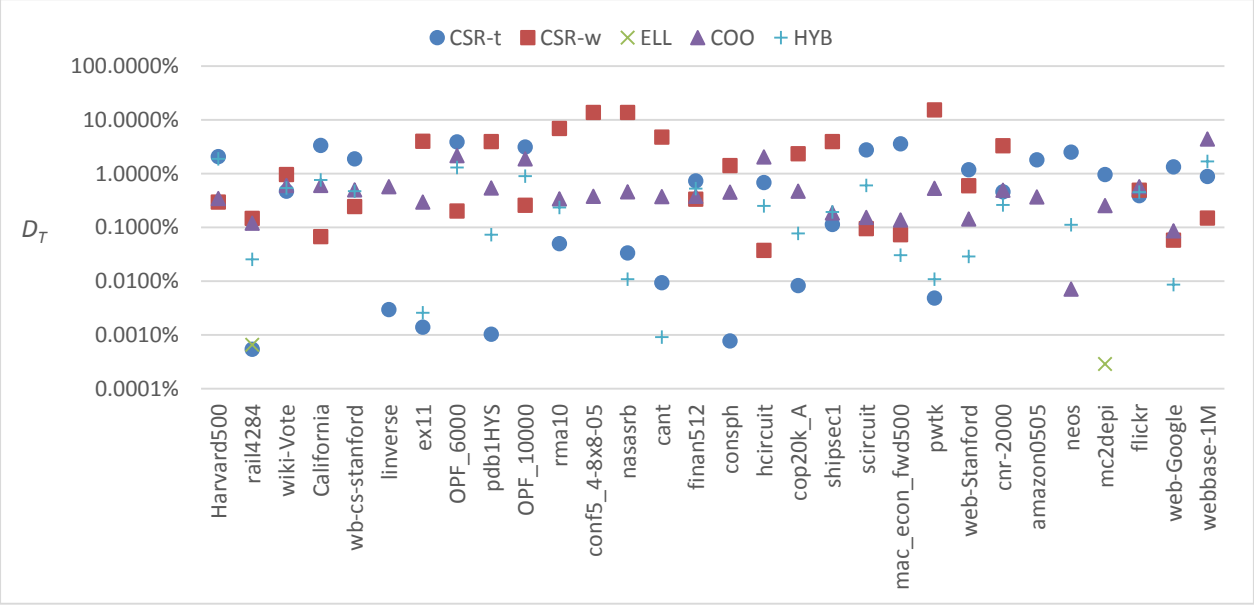


Figure 4.38 : Différences relatives entre les nombres de transactions estimés et mesurés pour les cinq implémentations du modèle

Les nombres de requêtes pour toutes les implémentations du modèle sont à la Figure 4.39. Cette fois-ci on remarque que c'est pour le format ELL qu'on observe très fréquemment les erreurs les plus importantes. La cause de ces erreurs est expliquée à la section 4.2.2. Pour le format COO, toutes les estimations sont exactes, aucun point ne se retrouve dans la figure. Toute comme pour le nombre de transactions, le nombre de requêtes du format CSR avec une chaîne par ligne (CSR-w) a été plus difficile à estimer, les erreurs présentent une valeur moyenne de 4.9 %. Pour les autres formats CSR-t et HYB, les erreurs sont plus élevées que pour les transactions, mais la moyenne reste encore faible avec 1.8 % et 1.48 % en moyenne respectivement.

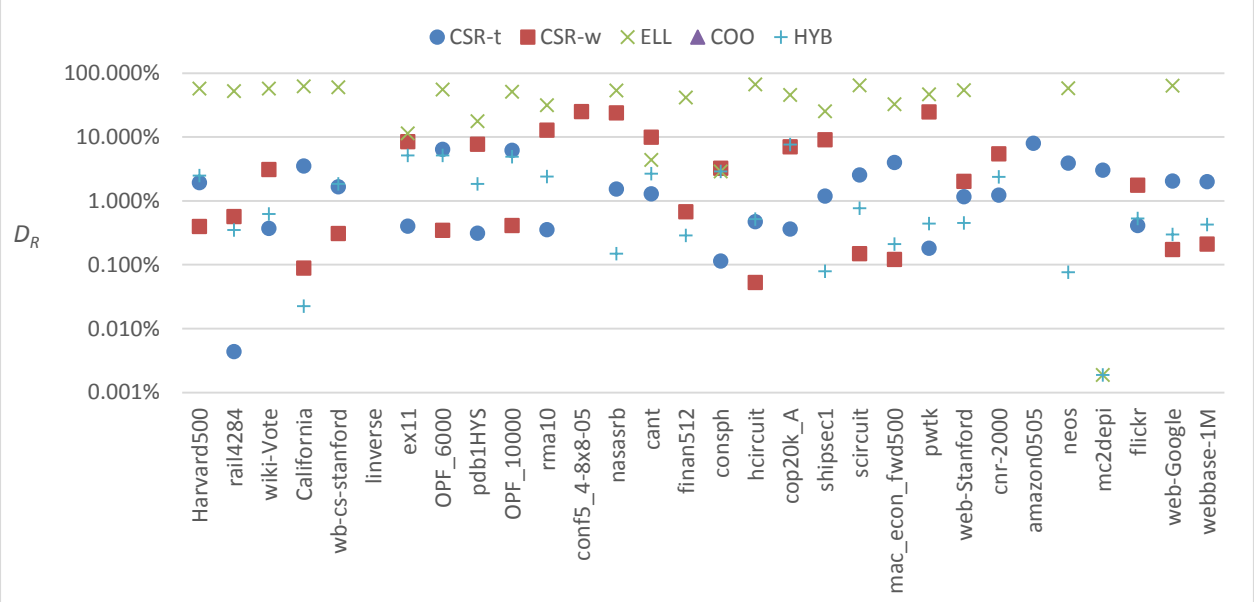


Figure 4.39 : Différences relatives entre les nombres de requêtes estimés et mesurés pour les cinq implémentations du modèle

Enfin, nous pouvons conclure que le modèle permet d'estimer les nombres de transactions et de requêtes mesurés avec une moyenne générale de 5 %.

4.3.2 Comparaison entre les formats

Le modèle proposé nous permet de comparer le taux de compression des formats, le nombre de transactions et de requêtes nécessaires à chaque format et le rapport des nombres de transactions par requête. Ainsi, le modèle proposé nous permet de mesurer la performance d'une implémentation de la SpMV par rapport à un autre en termes d'accès à la mémoire.

La comparaison entre la taille des matrices originales et la taille qu'elles ont lorsqu'elles sont formatées est présentée à la Figure 4.40. En général, le format CSR est celui qui compresse le plus, suivi de près par le format HYB. Comme attendu, le format ELL est celui qui compresse le moins la taille de la matrice originale. Pour les matrices *conf5_4-8x8-05* et *mc2depi*, le format ELL est mieux adapté puisque ces deux matrices ont des lignes ayant à peu près le même nombre d'éléments non nuls. Ceci fait en sorte que peu d'éléments nuls se retrouvent dans la représentation ELL et que la valeur de *K* est optimale.

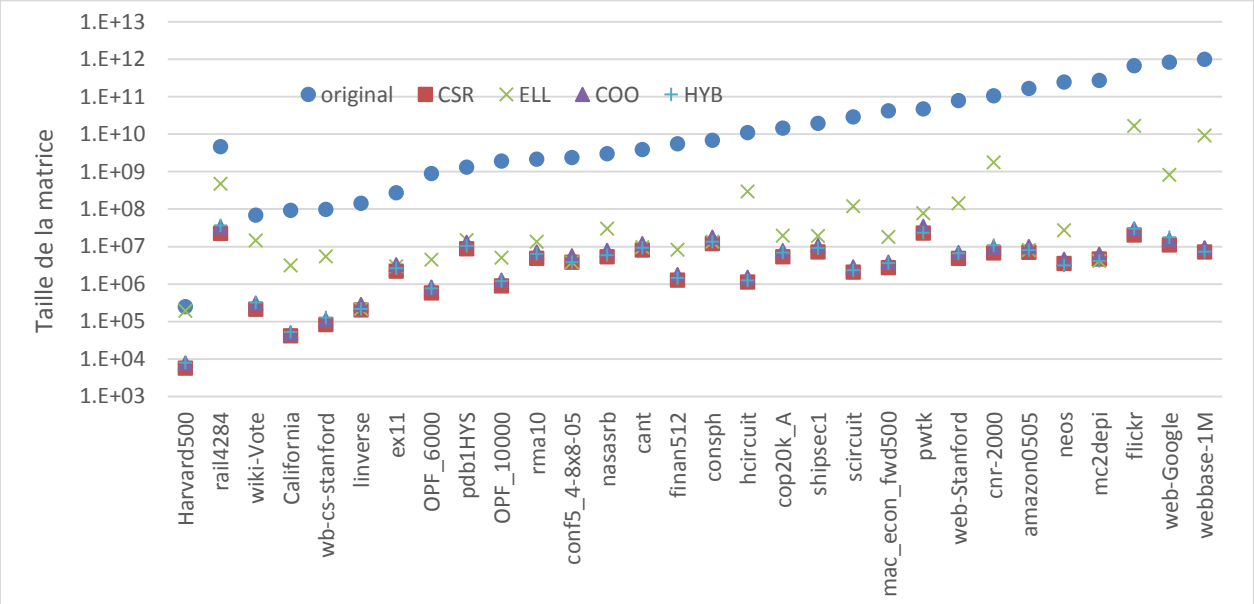


Figure 4.40 : Comparaison entre les tailles des matrices dans les différents formats de représentation

Une comparaison entre les nombres de transactions estimés des différentes implémentations est présentée à la Figure 4.41. On remarque que le nombre de transactions nécessaires pour effectuer une SpMV sur une matrice stockée en format CSR et implémentée avec un fil par ligne est l'un des plus élevés. L'implémentation utilisant le format CSR avec une chaîne par ligne réduit le nombre de transactions, mais reste quand même dans la moyenne. Le meilleur format selon le nombre de transactions est le format HYB, suivit du format COO qui le suit de près. Le format ELL, quant à lui, offre des nombres qui sont très changeants dépendamment du nombre de lignes et du nombre d'éléments par lignes. Le nombre de transactions dépend du nombre d'éléments qu'il est possible de transférer à la fois.

La comparaison des nombres de requêtes montre un tout autre résultat à la Figure 4.41. Le format HYB reste le meilleur format par rapport au nombre de requêtes, mais il est suivi de près par le

format CSR avec un fil d'exécution par ligne (CSR-t). Ainsi, l'implémentation CSR-t permet d'avoir moins de chaînes qui demandent des accès à la mémoire. Par contre, compte tenu du nombre de transactions qui a été mesuré, les éléments sont si dispersés que plusieurs transactions sont nécessaires pour répondre à chaque requête. Les autres formats ont des courbes semblables à leur courbe du nombre de transactions. Comme attendu, il y a, en général, moins de requêtes que de transactions. Le nombre de requêtes est important à comparer, car il est indépendant de la plateforme utilisée. Il dépend seulement du nombre d'instructions d'accès à la mémoire et du nombre de chaînes comme vu à l'équation (3.1). Ces nombres sont tirés de l'implémentation de la SpMV.

Un dernier élément important de comparaison entre différentes implémentations d'une même opération, en fonction des accès à la mémoire, est le nombre de transactions par requête. Cette comparaison est présentée à la Figure 4.43. Si le rapport est de 1, cela signifie que toutes les requêtes demandées n'ont besoin que d'une seule transaction à la mémoire. À partir des derniers résultats, on s'attend à ce que le format qui performe le moins bien dans cette catégorie soit le format CSR-t. On remarque aussi que les formats COO et CSR-w sont ceux qui requièrent le moins de transactions par requête avec une moyenne respective de 2 et 1.8 transaction par requêtes.

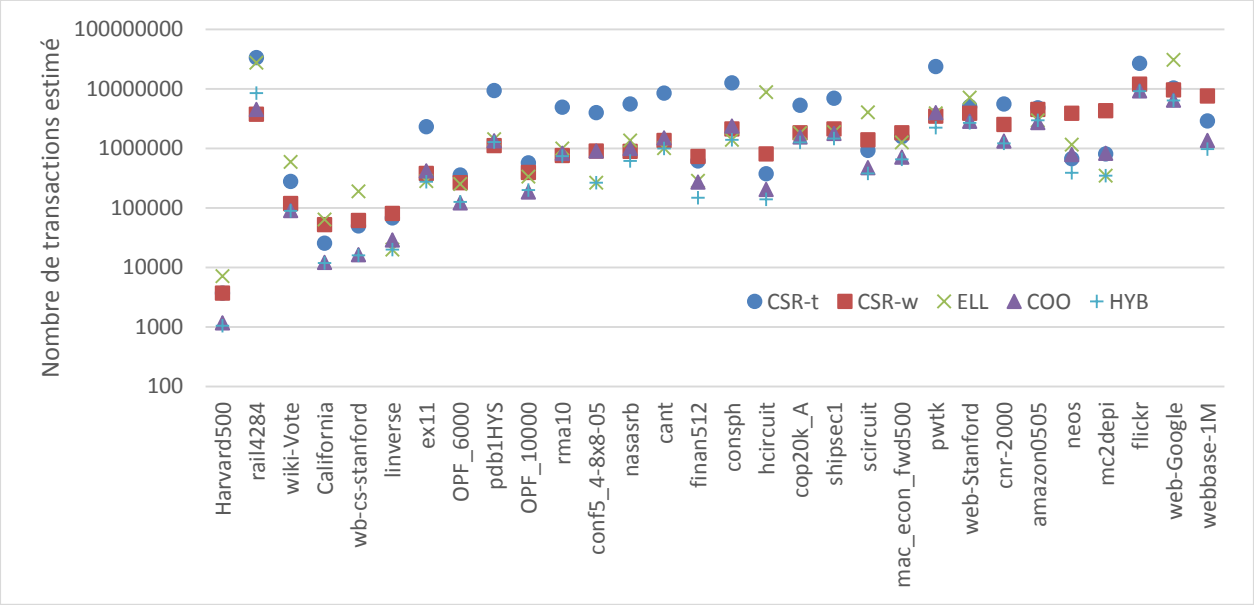


Figure 4.41 : Comparaison entre les nombres de transactions estimés avec chaque format

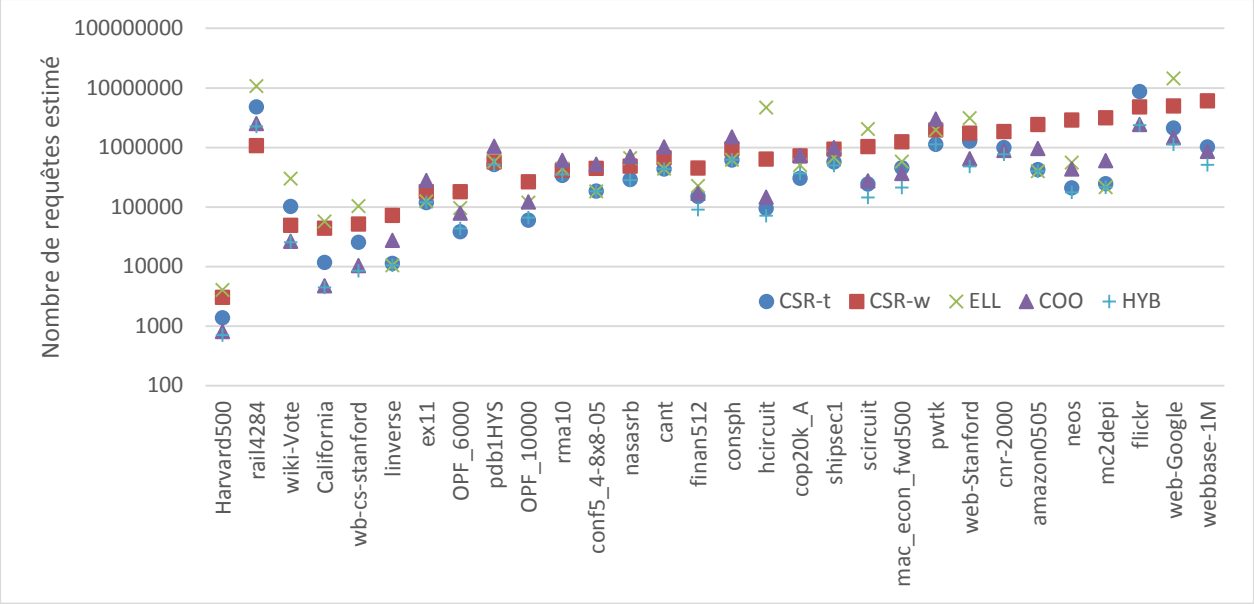


Figure 4.42 : Comparaison entre les nombres de requêtes estimés avec chaque format

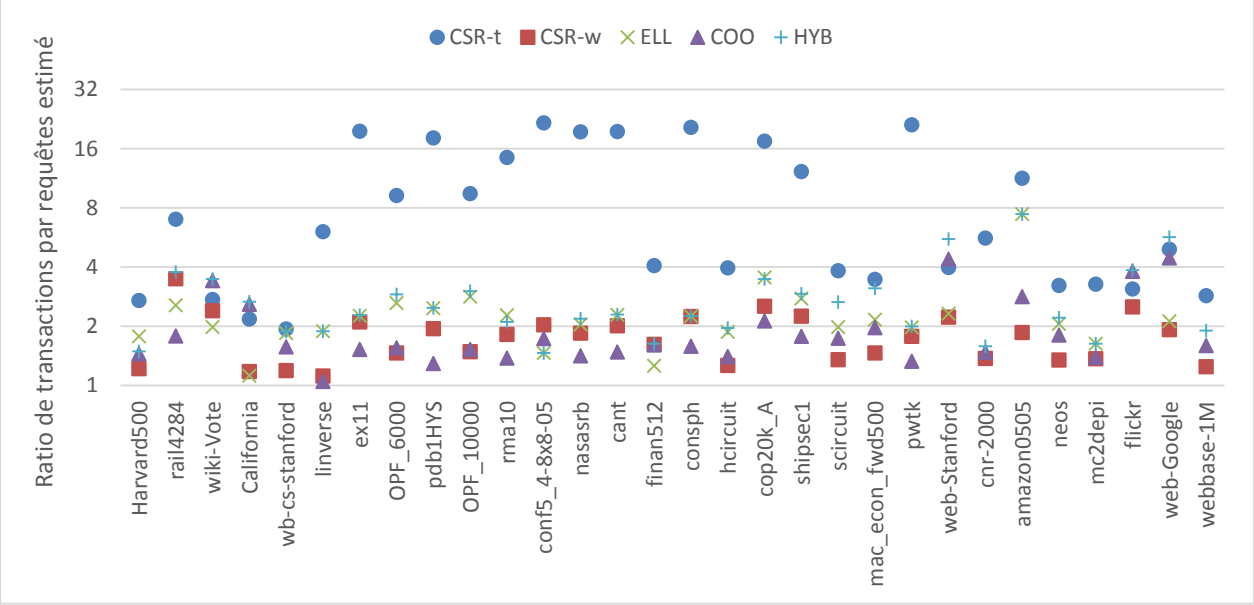


Figure 4.43 : Comparaison entre les nombres de transactions par requêtes estimés de chaque format

En résumé, le meilleur format en termes d'accès à la mémoire, pour l'ensemble des matrices testées, est le format HYB. Il permet de stocker une matrice avec peu d'éléments, effectue le moins de transactions et le moins de requêtes. De plus, puisque les accès à la mémoire sont une importante part du temps d'exécution dans l'implémentation de la SpMV, nous pouvons dire que le format HYB est aussi celui qui permet d'avoir les meilleurs temps d'exécutions pour notre ensemble de matrices creuses et les GPU utilisés. Pour le prouver, la Figure 4.44 présente une comparaison des

temps d'exécution de chaque format sur la GeForce GTX 670. L'implémentation du format HYB est la plus rapide pour 20 matrices sur 30. La plupart des matrices qui sont plus rapides avec une autre implémentation ont le plus petit nombre de lignes : *Harvard500*, *rail4284*, *wiki-Vote*, *California* et *wb-cs-stanford*. La section suivante discutera plus en détail du temps d'exécution.

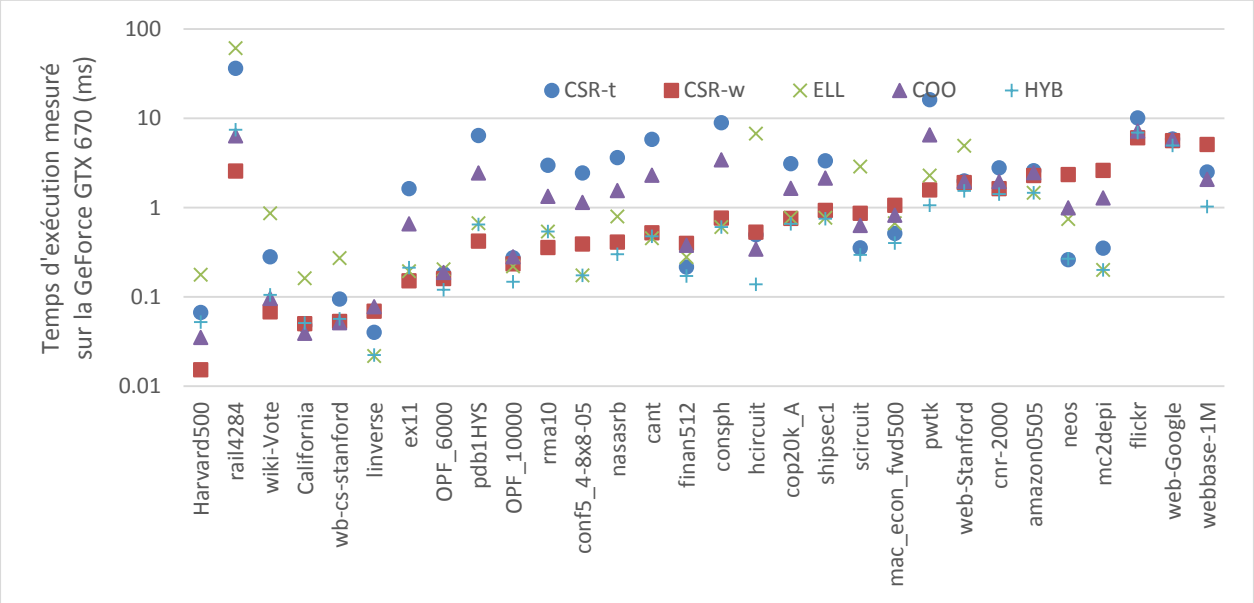


Figure 4.44 : Comparaison entre les temps d'exécution de chaque format sur la GeForce GTX 670

4.3.3 Évaluation du temps d'exécution

Le nombre de transactions à la mémoire calculé par le modèle permet d'évaluer la performance d'une implémentation de SpMV sur un GPU ayant une architecture Kepler. Puisque la performance de l'algorithme d'une SpMV est liée à la mémoire, le nombre de transactions permet de se faire une idée du temps d'exécution de l'algorithme.

Le temps d'exécution d'une implémentation peut être mesuré par les évènements CUDA. Ces évènements permettent de mesurer le temps d'exécution d'un noyau en millisecondes avec une précision de 0.5 microseconde. Tous les temps sont présentés à l'Annexe F et sont une moyenne sur 100 itérations. Les temps d'exécution de chaque implémentation sont présentés dans les figures suivantes : le format CSR avec un fil par ligne (CSR-t) à la Figure 4.45, le format CSR avec une chaîne par ligne (CSR-w) à la Figure 4.46, le format ELL à la Figure 4.47, le format COO à la Figure 4.48 et le format HYB à la Figure 4.49. Dans les figures, les temps sont présentés en nanosecondes afin de pouvoir les comparer au nombre de transactions.

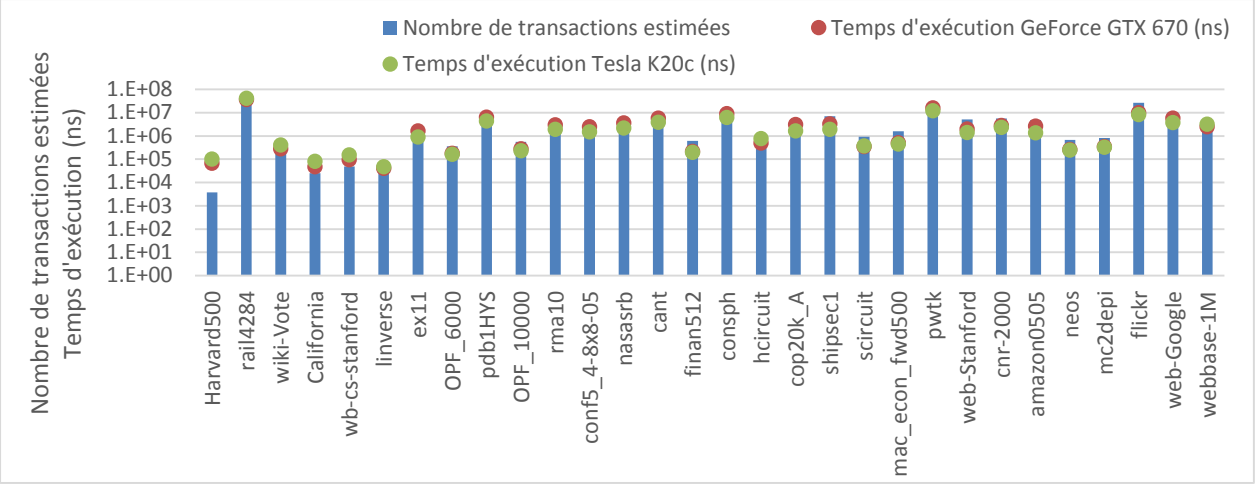


Figure 4.45 : Les nombres de transactions comparés au temps d'exécution pour le format CSR-t

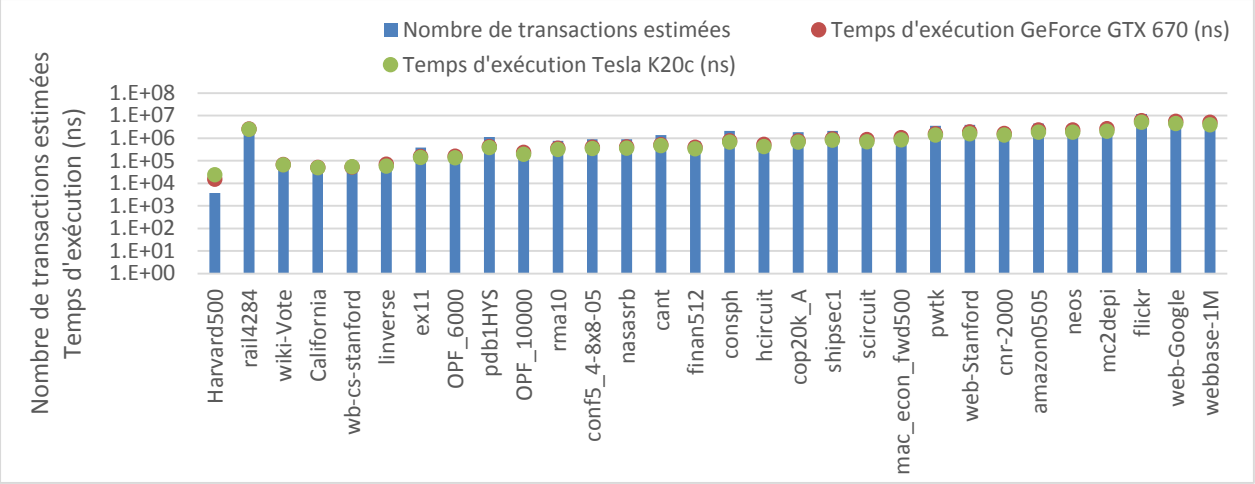


Figure 4.46 : Les nombres de transactions comparés au temps d'exécution pour le format CSR-w

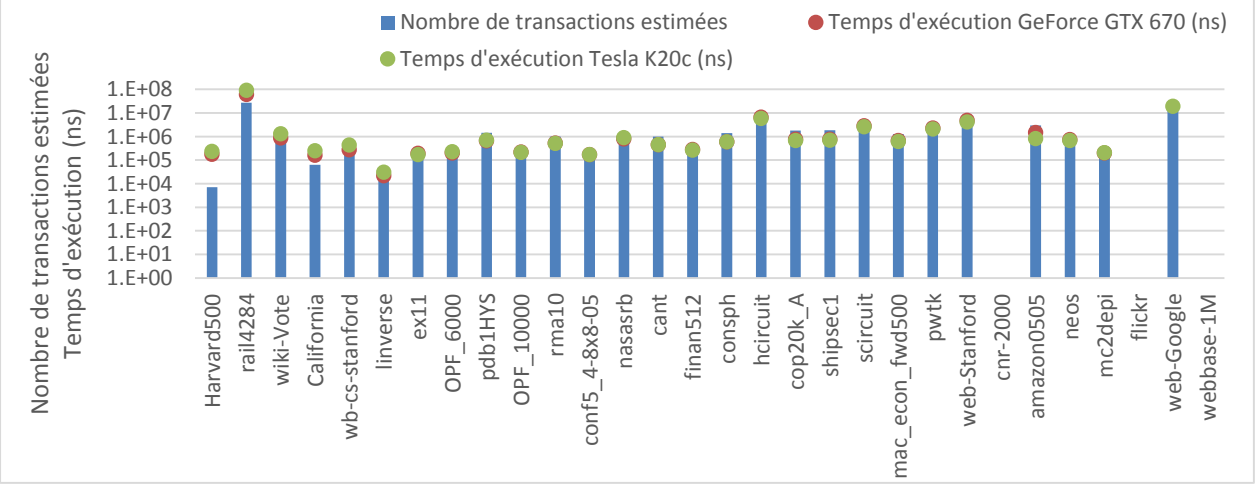


Figure 4.47 : Les nombres de transactions comparés au temps d'exécution pour le format ELL

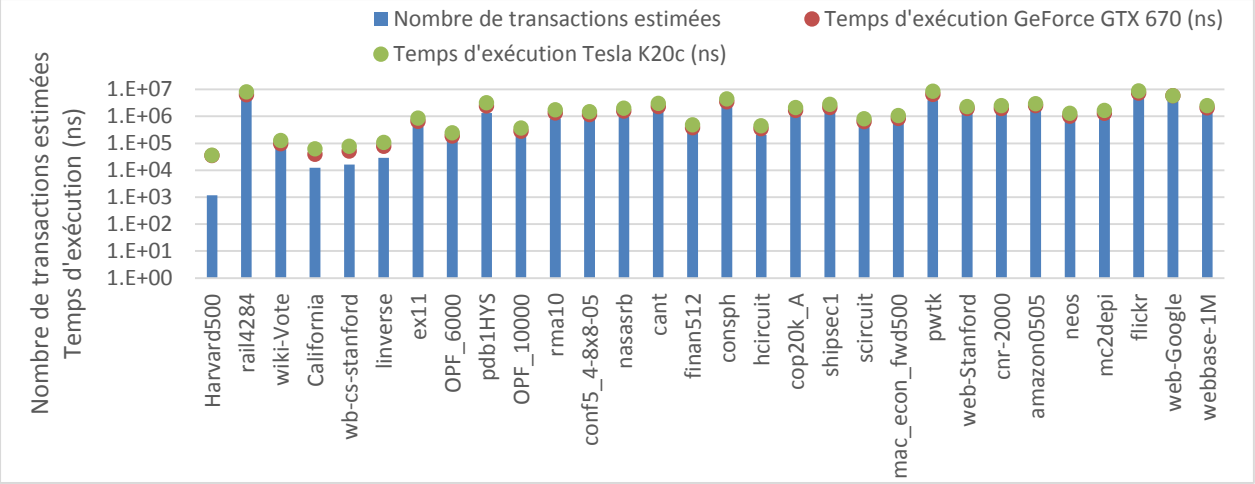


Figure 4.48 : Les nombres de transactions comparés au temps d'exécution pour le format COO

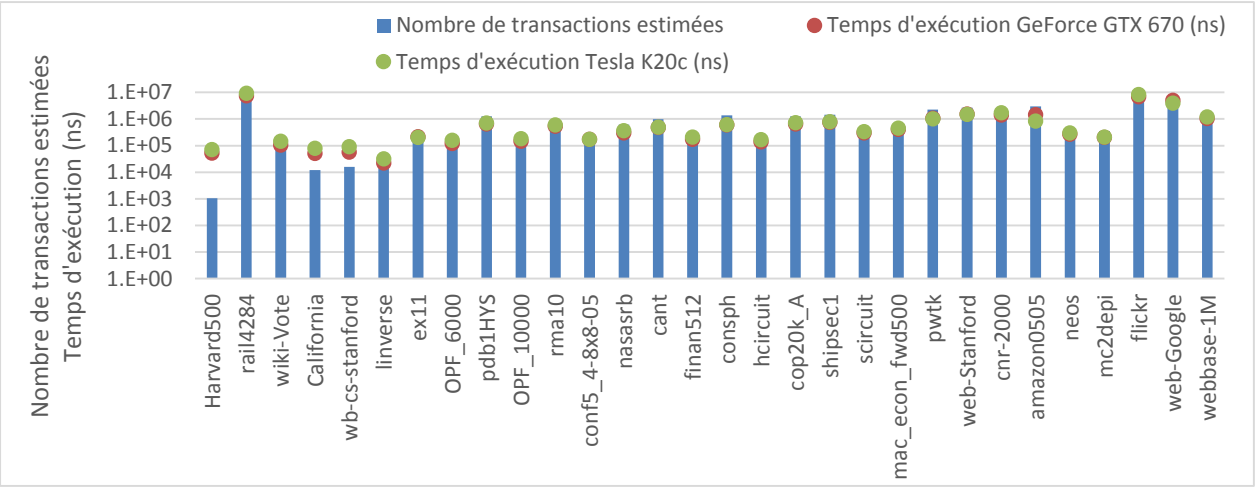


Figure 4.49 : Les nombres de transactions comparés au temps d'exécution pour le format HYB

On remarque dans les cinq figures de comparaison que le temps d'exécution est grandement influencé par le nombre de transactions. En effet, on peut voir que les temps d'exécution suivent les variations dans les nombres de transactions. Par contre, pour les matrices plus petites, de *Harvard500* à *linverse*, environ, le temps d'exécution est plus élevé que le nombre de transactions. Ceci est dû au fait que, puisque les matrices sont plus petites, toutes les régions du GPU ne sont pas utilisées à tout moment. Ainsi, il n'y a pas assez de parallélisation pour dissimuler complètement les temps d'accès à la mémoire, ce qui les rend plus longs. D'ailleurs ces matrices ont un faible taux d'occupation, une variable qui peut être calculée par le profileur.

Il n'y a pas une grande différence entre les temps d'exécution des deux GPU. Pour la majorité des matrices en format COO et HYB et près de la moitié des matrices en format ELL, le temps d'exécution sur la Tesla est plus élevé que le temps d'exécution sur la GeForce. Bien que la Tesla ait un plus grand nombre de cœurs permettant de réaliser plus d'instructions en parallèle, sa fréquence d'opération est moins élevée que celle de la GeForce. De plus, un plus grand nombre de cœurs n'est pas toujours avantageux, puisqu'il se peut aussi qu'il n'y ait pas assez de travail pour garder toutes les régions du GPU actives. Les avantages de choisir une Tesla, dans le contexte de ce projet, sont ses mémoires protégées contre les erreurs et de taille plus élevée. Si la précision n'est pas critique et que les matrices ciblées par notre application ne sont pas trop grandes, il ne semble pas y avoir un grand avantage à choisir une carte Tesla, surtout compte tenu de la grande différence de prix entre les deux cartes étudiées.

En supposant que le temps de calcul est totalement dissimulé par le temps d'accès à la mémoire, il est possible de créer une simple équation de premier ordre permettant de prédire le temps d'exécution (4.5). L'équation dépend du nombre de transactions, T, de la taille de transfert, S_{tx}, et de la bande passante, BP. Les GPU utilisés sont tous les deux de l'architecture Kepler et leurs transactions représentent des transferts de 128 octets à la fois. La bande passante maximale de chacun des GPU se calcule à partir de la fréquence de la mémoire f_{mem} et de la largeur du bus L (4.6). La fréquence est multipliée par deux ici, car la mémoire fonctionne avec un débit de données double (GDDR5). On divise le tout par 8 pour avoir une bande passante exprimée en octets par seconde.

$$t_{estimé} = \frac{T \times S_{tx}}{BP} \tag{4.5}$$

$$BP = \frac{f_{mem} \times 2 \times L}{8} \tag{4.6}$$

Évidemment, les temps d'exécution calculés à l'aide de cette équation ne sont pas exacts. Tous les temps estimés et les erreurs sont présentés à l'Annexe G. Pour la GeForce GTX 670, la moyenne des erreurs absolues entre le temps estimé et le temps moyen mesuré est de 38.6 %, 32.6 %, 32.8 %, 52.1 % et 133 % pour les formats CSR-t, CSR-w, ELL, COO, HYB respectivement. Dans le même ordre, pour la Tesla K20c, la moyenne des erreurs est de 66.4 %, 43.0 %, 36.8 %, 64.8 % et 143.8 %. Ainsi, bien que les nombres de transactions à la mémoire soient un facteur important qui affecte les temps d'exécution, ce n'est pas le seul facteur.

Par contre, pour certaines matrices, le temps d'exécution d'une implémentation est estimé avec peu d'erreurs. C'est le cas, par exemple, de la matrice *ex11* pour le format HYB avec 0.28 % d'erreur entre le temps d'exécution estimé et mesuré sur la GeForce GTX 670. D'ailleurs, dans ce cas, le nombre de transactions est parfaitement estimé par le modèle. Le temps d'exécution de l'implémentation de la SpMV pour cette matrice en format HYB dépend totalement des accès à la mémoire. Tout temps passé à faire des calculs est dissimulé par un accès à la mémoire en parallèle. Pour d'autres matrices, l'erreur est très élevée. La matrice *flickr* atteint une erreur d'environ 1500 % pour le format HYB sur la GeForce. Dans ce cas, beaucoup de temps est passé à réaliser d'autres instructions que des instructions de mémoire, comme le calcul de l'adresse de mémoire.

Ainsi, pour estimer le temps d'exécution, une simple équation de premier ordre n'est pas assez précise. Par contre, le nombre de transactions et une estimation de premier ordre du temps d'exécution peuvent permettre de situer une implémentation par rapport à d'autres et de choisir l'implémentation et le format qui est le plus adapté pour une matrice.

4.4 Comparaison avec la littérature

Le modèle présenté dans ce mémoire permet de prédire les accès à la mémoire d'une SpMV sur un processeur graphique. D'autres auteurs présentés lors de la revue de littérature ont aussi réalisé des modèles sur les GPU. Le temps d'exécution est mesuré par plusieurs auteurs en utilisant un modèle analytique. C'est le cas de Hong et Kim [44] avec un modèle des processeurs graphiques en général. Le nombre de transactions est utilisé dans leur modèle, par contre, il ne semble pas y avoir de cas particulier où ce nombre est difficile à estimer. Dans le cas de la SpMV, comme le démontre ce mémoire, le nombre de transactions est difficile à estimer. Ce sujet n'est malheureusement pas abordé dans leur article. Le modèle proposé pourrait donc compléter le modèle de Hong et Kim lorsque l'implémentation à estimer doit accéder à la mémoire de façon irrégulière. Pour la SpMV en particulier, Choi et al. [43] ont présenté un modèle analytique, El Zein et Rendell [46] un modèle expérimental et Guo et al. [47] un modèle qui utilise des techniques analytiques et expérimentales. Ces modèles se concentrent sur la prédiction du temps d'exécution alors que le modèle proposé dans ce mémoire se base sur le nombre de transactions à la mémoire. Le modèle proposé présente une méthode alternative de prédiction de la performance de la SpMV. Le nombre d'accès à la mémoire prédit par notre modèle a une erreur moyenne de 5 %, ce qui est comparable aux précisions des modèles de la littérature. Par exemple, la précision du modèle de Guo et al varie

entre 0 et 10 % pour les mêmes formats CSR, ELL, COO et HYB avec un ensemble de matrices testées comparable.

Le modèle présenté dans ce mémoire permet aussi de comparer différentes implémentations de la SpMV. Les expérimentations de Bell et Garland [18] comparent les deux implémentations du format CSR, le format COO et le format HYB sur un ensemble de matrices. Ces matrices font partie de l'ensemble de matrices utilisées dans ce projet. Ils démontrent que, pour des matrices composées de nombres à virgule flottante à simple précision, le format HYB est le meilleur format, puisqu'il permet d'avoir le plus haut nombre d'opérations à la seconde et la plus haute bande passante. À l'opposé, c'est le format CSR avec un fil par ligne qui a la performance la plus faible par rapport à ces deux métriques. Les résultats de notre modèle permettent d'arriver aux mêmes conclusions en comparant simplement le nombre total de transactions et de requêtes.

4.5 Améliorations possibles

Suite à la réalisation du modèle proposé et de ses résultats, plusieurs améliorations possibles se manifestent. Nous présenterons des améliorations du format ELL, COO et HYB qui découlent directement de l'analyse de leur implémentation réalisée dans le Chapitre 3, ainsi qu'une piste sur l'utilisation de différentes régions de la mémoire sur les GPU.

En analysant l'implémentation de la SpMV utilisant le format ELL, nous avons observé que le nombre de transactions augmente en fonction du nombre de lignes de la matrice. Si le nombre de lignes total de la matrice n'est pas un multiple du nombre d'éléments par transfert, plusieurs transactions superflues sont réalisées. Les adresses demandées, bien que consécutives, seront décalées par rapport à l'alignement et demanderont plus d'un transfert. Il conviendrait alors d'ajouter des lignes de zéro à la représentation ELL afin que le nombre de lignes soit un multiple de S_{tx}/S_e.

Par exemple, à la Figure 4.50, une matrice de sept lignes et trois colonnes est représentée. À gauche, la matrice est représentée de façon traditionnelle. À droite, elle est représentée en mémoire, en ordre de colonnes et avec quatre colonnes représentant une taille de transfert de quatre éléments de matrices. Lorsqu'une requête est réalisée pour lire un élément de la matrice, les quatre éléments de la ligne à laquelle il se trouve seront transférés.

Lorsque l'implémentation de la SpMV sera arrivée à traiter la multiplication du deuxième élément de chaque ligne avec le vecteur, les fils d'exécution demanderont en mémoire les valeurs (0,1) à

(6,1). Pour réaliser cette lecture, trois transferts de quatre éléments seront effectués. Le premier transfert représente la deuxième ligne de la matrice en mémoire contenant l'élément (0,1). Le deuxième transfert représente les éléments (1,1) à (4,1). Le troisième transfert représente les quatre éléments de la troisième ligne de la matrice en mémoire qui contient les éléments (5,1) et (6,1). Pour effectuer la lecture de toute la matrice au complet, le nombre de transactions sera de 8.

Par contre, si la matrice originale est modifiée afin d'avoir 8 lignes, un multiple de quatre, le nombre de transactions en sera diminué. Ce changement est démontré à la Figure 4.51, où une ligne de 0 est ajoutée à la matrice de gauche. Si l'on demande la lecture des éléments (0,1) à (6,1) et du 0, deux transferts seront réalisés. En effet, avec la configuration en mémoire de la matrice à droite, nous remarquons que la première adresse demandée est alignée et aucun élément superflu n'est transféré afin de répondre à la requête de lecture. Dans ce cas-ci, le nombre total de transactions pour lire la matrice est de 6, même si nous avons y ajouté des éléments.

(0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)
(2,0)	(2,1)	(2,2)
(3,0)	(3,1)	(3,2)
(4,0)	(4,1)	(4,2)
(5,0)	(5,1)	(5,2)
(6,0)	(6,1)	(6,2)

(a)

(0,0)	(1,0)	(2,0)	(3,0)
(4,0)	(5,0)	(6,0)	(0,1)
(1,1)	(2,1)	(3,1)	(4,1)
(5,1)	(6,1)	(0,2)	(1,2)
(2,2)	(3,2)	(4,2)	(5,2)
(6,2)			

(b)

Figure 4.50 : Exemple d'une matrice en format ELL à gauche (a) et la façon dont elle est mise en mémoire avec S_w égal à quatre à droite (b)

(0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)
(2,0)	(2,1)	(2,2)
(3,0)	(3,1)	(3,2)
(4,0)	(4,1)	(4,2)
(5,0)	(5,1)	(5,2)
(6,0)	(6,1)	(6,2)
0	0	0

(a)

(0,0)	(1,0)	(2,0)	(3,0)
(4,0)	(5,0)	(6,0)	0
(0,1)	(1,1)	(2,1)	(3,1)
(4,1)	(5,1)	(6,1)	0
(0,2)	(1,2)	(2,2)	(3,2)
(4,2)	(5,2)	(6,2)	0

(b)

Figure 4.51 : Exemple d'une matrice en format ELL avec ligne de zéros à gauche (a) et la façon dont elle est mise en mémoire avec S_w égal à quatre à droite (b)

L'analyse de l'implémentation utilisant le format COO peut être améliorée. En effet, l'intervalle qui a été utilisé dans le modèle est de taille S_w. Cela signifie que chaque chaîne s'occupe de S_w éléments de la matrice formatée, ce qui revient à dire que chaque fil d'exécution s'occupe de multiplier un seul élément non nul. Par contre, la grandeur de l'intervalle n'est pas fixe dans l'implémentation de la bibliothèque sur laquelle est basé le modèle. L'implémentation originale permet de choisir une longueur d'intervalle selon un nombre maximal de blocs qui dépend du GPU utilisé. Si une valeur d'intervalle de S_w appelle trop de blocs, l'algorithme divise l'opération en plusieurs itérations en augmentant l'intervalle et en assignant plus d'éléments non nuls par chaîne. Il serait donc intéressant d'observer le comportement de plusieurs grandeurs d'intervalle et de caractériser son influence sur le nombre de transactions, de requêtes ou du temps d'exécution.

Pour le format HYB, en plus d'analyser l'optimisation de la section ELL en ajoutant des lignes de zéros à sa représentation, il serait intéressant de parler de la valeur optimale de K. En effet, dans ce mémoire, la valeur de K est choisie selon la définition de Bell et Garland [18] où le nombre optimal est atteint lorsqu'au moins un tiers des lignes de la matrice contient K ou plus éléments non nuls. Par contre, dans la bibliothèque CUSP [14], le nombre optimal K est différent. Il serait donc intéressant d'ajouter au modèle la possibilité de modifier la valeur de K optimale afin d'observer l'impact qu'il a sur les nombres de transactions ou de requêtes. Ainsi, il serait possible de calculer la valeur de K optimal selon la distribution des éléments non nuls dans la matrice cible et selon les caractéristiques du GPU cible.

Enfin, pour toutes les implémentations présentées, le modèle prenait en compte le fait que la matrice était stockée dans la mémoire globale. Il arrivait quelques fois d'utiliser la mémoire partagée, mais seulement pour y entreposer des valeurs temporaires qui devaient être utilisées par plusieurs fils d'exécutions. D'importantes observations pourraient être réalisées si d'autres espaces de mémoire étaient utilisés. Par exemple, si la matrice n'a pas trop de colonnes, il serait possible de stocker le vecteur x dans la mémoire partagée, la mémoire de constantes ou de texture. Puisqu'un élément du vecteur x peut être accédé par différents fils, cela permettrait de réduire le temps d'accès à la mémoire pour ces requêtes.

CHAPITRE 5 CONCLUSION

Le sujet de ce mémoire découle tout d'abord d'un désir d'accélérer une application sur GPU. Dans l'optique d'utiliser les GPU dans des applications de centres de données, nous nous sommes d'abord tournés vers les algorithmes de moteurs de recherche. L'algorithme PageRank fut l'un des algorithmes observés. Une première implémentation de celui-ci sur GPU nous a démontré que, bien que les GPU ont des milliers de cœurs, l'accélération de l'algorithme PageRank n'était qu'environ 2 fois plus rapide sur GPU que sur CPU. L'opération la plus importante de l'algorithme PageRank est la multiplication d'une matrice creuse, représentant les interconnexions entre les pages web et un vecteur, représentant l'importance de chacune de celles-ci.

Un des facteurs les plus importants qui auraient pu empêcher les GPU de performer autant qu'en théorie est les accès à la mémoire. En effet, une différence importante entre les CPU et les GPU est la façon dont la mémoire est accédée. Un CPU est réalisé pour surtout réduire la latence des opérations qui sont effectuées. Ainsi, il possède trois étages de cache ayant des tailles assez grandes permettant de réduire les temps d'accès à la mémoire. Sur les GPU, les bandes passantes de chaque région de la mémoire sont très élevées, mais les tailles inférieures des transferts et des caches font toute la différence. Des accès irréguliers à la mémoire des GPU peuvent ralentir énormément un algorithme. De plus, l'opération de multiplication entre une matrice creuse et un vecteur avait été mainte fois, dans la littérature, qualifiée de difficile à optimiser à cause de ses accès irréguliers à la mémoire.

Ainsi, l'objectif principal de ce mémoire a été d'expliquer la raison derrière les grands temps d'accès à la mémoire de la SpMV. Ceci nous a donc amené à réaliser une analyse des GPU sur les façons optimales d'accéder à leur mémoire. Nous avons aussi analysé différentes implémentations de la SpMV afin de trouver leurs avantages et désavantages. Enfin, la nécessité d'un modèle s'est fait sentir, car la prédiction de la performance d'une implémentation par une autre n'est pas une tâche banale. La nature irrégulière des accès à la mémoire ainsi que la grande taille des matrices rendaient les choses plus difficiles.

La prochaine section présente une synthèse des contributions. Nous y verrons une description du modèle, de ce qu'il permet d'estimer et des résultats obtenus. Puis, nous verrons les travaux futurs que ce projet nous permet de faire.

5.1 Synthèse des contributions

Dans ce mémoire, nous avons proposé un modèle permettant de mesurer le nombre de transactions et de requêtes à la mémoire pour la multiplication d'une matrice creuse par un vecteur lorsqu'elle est implémentée sur un processeur graphique. Cette opération est difficile à optimiser puisque les matrices sont de très grandes tailles, leurs éléments ne suivent pas toujours une structure prédéfinie et les accès à la mémoire sont irréguliers. Ainsi, plusieurs implémentations sont proposées afin de permettre à cette opération de prendre moins de temps. Le modèle proposé permet de comparer différentes implémentations entre elles et de permettre aux programmeurs de faire un choix éclairé sur la meilleure implémentation selon leurs besoins.

Le modèle proposé prend en compte plusieurs paramètres dont la représentation de la matrice dans un format, les caractéristiques du GPU utilisé et l'implémentation qui sera utilisée dans le noyau. Quatre formats appelés CSR, ELL, COO et HYB ont été choisis pour appliquer notre modèle. Ces formats sont utilisés par deux bibliothèques de manipulation de matrices creuses qu'il est possible d'utiliser avec le langage CUDA. Seuls les GPU de la société NVIDIA peuvent être programmés en utilisant ce langage CUDA. Le modèle prend en compte les versions 1.0 à 3.5 des GPU NVIDIA. Chacun des formats considérés est relié à une implémentation qui démontre la façon dont les tâches sont distribuées entre les fils d'exécution. Le format CSR a fait l'objet de deux implémentations, l'une où chaque fil s'occupe de multiplier une ligne de la matrice avec le vecteur et un autre où chaque ligne de la matrice est gérée par une chaîne de fils.

Les résultats produits par le modèle sont le nombre de transactions et de requêtes de chaque implémentation. Une requête représente une demande d'accès à la mémoire par une chaîne de fils et une transaction représente un transfert de données faisant partie de cette requête. Le nombre de requêtes dépend de l'implémentation du noyau et de la distribution des tâches sur les fils d'exécutions. Le nombre de transactions dépend du processeur utilisé et de la façon dont il communique avec sa mémoire, ainsi que de la représentation de la matrice creuse dans la mémoire. Dépendamment du format et de l'implémentation, une requête peut demander d'une à plusieurs transactions. Plus une requête demande de transactions et moins l'implémentation choisie est idéale pour la matrice en question.

Des tests ont été réalisés sur un ensemble de matrices creuses non structurées provenant de plusieurs domaines. Chaque implémentation a été exécutée sur deux cartes GPU : la GeForce GTX

670 et la Tesla K20c. Les nombres de transactions et de requêtes ont été mesurés grâce à un profilage de chaque implémentation sur chacune des matrices. Les résultats démontrent que le nombre de transactions et de requêtes estimé est très près de la réalité. La moyenne générale de l'erreur relative de toutes les implémentations confondues est de 5.1 %. Les moyennes d'erreurs relatives des nombres de transactions sont respectivement de 1.1 %, 2.6 %, 0 %, 0.7 % et 0.3 % lorsque l'implémentation utilise le format CSR avec un fil par ligne, le format CSR avec une chaîne par ligne, le format ELL, le format COO et le format HYB. Lorsque le modèle est utilisé pour estimer le nombre de requêtes, les moyennes d'erreurs pour les cinq implémentations sont respectivement de 1.8 %, 5.2 %, 37.5 %, 0 % et 1.5 %.

Les nombres de transactions et de requêtes estimés permettent de comparer les formats et d'évaluer leurs performances pour chaque matrice testée. Les résultats démontrent que le format présentant le moins de transactions pour la majorité des matrices est le format HYB. Le format demandant le moins de requêtes à la mémoire est aussi le format HYB. Il est possible de comparer les formats selon le nombre de transactions par requête. En moyenne, les cinq implémentations ont respectivement un nombre de transactions par requête de 9.3, 1.8, 2.3, 2 et 2.8. Selon cette métrique, le meilleur format est le format CSR avec une chaîne par ligne.

5.2 Travaux futurs

Le modèle proposé a été réalisé pour des implémentations simples des formats. Des travaux futurs pourraient se concentrer sur l'optimisation de ces implémentations et la caractérisation de l'impact que cela aurait sur le nombre d'accès à la mémoire. Plusieurs améliorations ont été présentées à la section 4.5. Il est possible d'améliorer les implémentations utilisant les formats ELL, COO et HYB qui ont été présentées dans ce mémoire. La section discute aussi de la pertinence d'utiliser d'autres régions de la mémoire des GPU.

Une meilleure estimation du temps d'exécution pourrait être réalisée afin de le prédire avec précision. Dans ce mémoire, une simple estimation de premier ordre est réalisée en utilisant le nombre de transactions et la bande passante théorique. Par contre la bande passante n'est pas fixe sur les GPU, elle dépend de la façon dont le travail est partagé entre les fils et de la possibilité de réaliser des accès coalescents à la mémoire. Lorsque les accès demandent des adresses consécutives et

alignées, la bande passante est maximale. Le temps d'exécution devrait prendre en compte la performance de l'implémentation testée en fonction du nombre de transactions par requête pour évaluer la performance de la bande passante.

De plus, une deuxième partie du calcul devrait observer si l'occupation du GPU est totale et si les opérations de calculs peuvent être complètement dissimulées. Dépendamment de ce nombre, un délai pourrait être ajouté pour les calculs arithmétiques qui ne peuvent être dissimulés par un transfert de données en mémoire.

D'autres travaux pourraient améliorer le présent projet. L'un d'entre eux serait d'étendre l'ensemble des matrices afin d'inclure des matrices d'autres types; des matrices complexes, des nombres à virgule flottante de double précision ou des nombres entiers. De plus, les matrices utilisées pourraient inclure des matrices structurées telles que des matrices diagonales où à plusieurs diagonales. Le nombre de fils d'exécution par bloc pourrait être paramétrable afin de trouver la meilleure distribution des tâches. Plus de formats pourraient être modélisés ainsi qu'un nouveau format pourrait être proposé qui permettrait d'atteindre les meilleurs résultats dans chaque catégorie.

RÉFÉRENCES

[1] R. P. Tewarson, *Sparse Matrices*, Academic Press, 1973, p. 160.

[2] A. Brameller, “Sparsity in Transportation Problems,” in *Sparsity and Its Applications*, CUP Archive, 1985, pp. 229–242.

[3] L. Page, S. Brin, R. Motwani, and T. Winograd, “The PageRank Citation Ranking: Bringing Order to the Web,” Stanford InfoLab, Rapport technique SIDL-WP-1999-0120, 1998.

[4] F. Steidler, “Sparsity applications in geodesy and photogrammetry,” in *Sparsity and Its Applications*, 1985, pp. 303–320.

[5] M. L. Crow, “Sparse Matrix Solution Techniques,” in *Computational Methods for Electric Power Systems*, CRC Press, 2002, p. 81.

[6] S. Pissanetzky, *Sparse Matrix Technology*, Academic Press, 1984, p. 336.

[7] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, Society for Industrial and Applied Mathematics, 1994.

[8] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, “A Survey of General-Purpose Computation on Graphics Hardware,” *Comput. Graph. Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[9] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, “GPU Computing,” in *Proceedings of the IEEE*, vol. 96, no. 5, 2008, pp. 879–899.

[10] D. R. Kincaid, J. R. Respess, D. M. Young, and R. R. Grimes, “Algorithm 586: ITPACK 2C: A FORTRAN Package for Solving Large Sparse Linear Systems by Adaptive Accelerated Iterative Methods,” *ACM Transactions on Mathematical Software*, vol. 8, no. 3, pp. 302–322, 1982.

[11] T. C. Oppe, W. D. Joubert, and D. R. Kincaid, “An overview of NSPCG: A nonsymmetric preconditioned conjugate gradient package,” *Computer Physics Communications*, vol. 53, no. 1–3, pp. 283–293, 1989.

[12] Y. Saad, “SPARSKIT: a basic tool kit for sparse matrix computations,” 1994.

[13] NVIDIA, “cuSPARSE Library,” *NVIDIA CUDA Zone*, 2010. [En ligne]. Disponible: https://developer.nvidia.com/cusparse. [Consulté le 31 mai 2013].

[14] N. Bell and M. Garland, “Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations,” 2012. [En ligne]. Disponible: http://cusp-library.googlecode.com. [Consulté le 31 mai 2013].

[15] T. Davis, “University of Florida Sparse Matrix Collection.” [En ligne]. Disponible: http://www.cise.ufl.edu/research/sparse/matrices/. [Consulté le 27 mars 2013].

[16] R. F. Boisvert, R. Pozo, and K. A. Remington, “The Matrix Market Exchange Formats : Initial Design,” *NISTIR*, 1996.

[17] R. F. Boisvert, R. Pozo, K. Remington, R. F. Barrett, and J. J. Dongarra, “Matrix Market.” [En ligne]. Disponible: http://math.nist.gov/MatrixMarket/. [Consulté le 16 septembre 2013].

[18] N. Bell and M. Garland, “Efficient Sparse Matrix-Vector Multiplication on CUDA,” NVIDIA, Rapport technique NVR-2008-004, 2008.

[19] F. Vázquez, J. J. Fernández, and E. M. Garzón, “A new approach for sparse matrix vector product on NVIDIA GPUs,” *Concurrency and Computation: Practice and Experience*, vol. 23, no. 8, pp. 815–826, 2011.

[20] G. E. Blelloch, M. A. Heroux, and M. Zagha, “Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors,” School of Computer Science, Carnegie Mellon University, Rapport technique CMU-CS-93-173, 1993.

[21] S. Brin and L. Page, “The anatomy of a large-scale hypertextual Web search engine,” *Computer Networks and ISDN Systems*, vol. 30, no. 1–7, pp. 107–117, 1998.

[22] NVIDIA, “CUDA Toolkit Documentation.” *NVIDIA Developer Zone*, [En ligne]. Disponible: http://docs.nvidia.com/cuda/index.html. [Consulté le 10 juillet 2013].

[23] Khronos, “OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems.” [En ligne]. Disponible: http://www.khronos.org/opencl/. [Consulté le 29 août 2013].

[24] AMD, "OpenCL™ Zone." *AMD Developer Central*, [En ligne]. Disponible: http://developer.amd.com/tools-and-sdks/opencl-zone/. [Consulté le 22 septembre 2014].

[25] K. Karimi, N. G. Dickson, and F. Hamze, "A Performance Comparison of CUDA and OpenCL," *Computing Research Repository*, p. 12, 2010.

[26] S. Toledo, "Improving the memory-system performance of sparse-matrix vector multiplication," *IBM Journal of Research and Development*, vol. 41, no. 6, pp. 711–725, 1997.

[27] A. Pinar and M. T. Heath, "Improving performance of sparse matrix-vector multiplication," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, 1999.

[28] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization Framework for Sparse Matrix Kernels," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 135–158, 2004.

[29] J. B. White and P. Sadayappan, "On improving the performance of sparse matrix-vector multiplication," in *Proceedings Fourth International Conference on High-Performance Computing*, 1997, pp. 66–71.

[30] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007.

[31] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, 2013, p. 273.

[32] M. deLorimier and A. DeHon, "Floating-point sparse matrix-vector multiply for FPGAs," in *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, 2005, p. 75.

[33] L. Zhuo and V. K. Prasanna, "Sparse Matrix-Vector multiplication on FPGAs," in *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, 2005, p. 63.

[34] S. McGettrick, D. Geraghty, and C. McElroy, “An FPGA architecture for the Pagerank eigenvector problem,” in *2008 International Conference on Field Programmable Logic and Applications*, 2008, pp. 523–526.

[35] Y. Zhang, Y. H. Shalabi, R. Jain, K. K. Nagar, and J. D. Bakos, “FPGA vs. GPU for sparse matrix vector multiply,” in *2009 International Conference on Field-Programmable Technology*, 2009, pp. 255–262.

[36] M. M. Baskaran and R. Bordawekar, “Optimizing sparse matrix-vector multiplication on GPUs,” IBM Research, Rapport technique RC24704, 2009.

[37] T. Wu, B. Wang, Y. Shan, F. Yan, Y. Wang, and N. Xu, “Efficient PageRank and SpMV Computation on AMD GPUs,” in *Proceedings of the 39th International Conference on Parallel Processing*, 2010, pp. 81–89.

[38] F. Vázquez, G. Ortega, J. J. Fernández, and E. M. Garzón, “Improving the Performance of the Sparse Matrix Vector Product with GPUs,” in *2010 10th IEEE International Conference on Computer and Information Technology*, 2010, pp. 1146–1151.

[39] A. Monakov, A. Lokhmotov, and A. Avetisyan, “Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures,” in *High Performance Embedded Architectures and Compilers*, 2010, pp. 111–125.

[40] A. Dziekonski, A. Lamecki, and M. Mrozowski, “A Memory Efficient and Fast Sparse Matrix Vector Product on a GPU,” *Progress In Electromagnetics Research*, vol. 116, pp. 49–63, 2011.

[41] H.-V. Dang and B. Schmidt, “CUDA-enabled Sparse Matrix–Vector Multiplication on GPUs using atomic operations,” *Parallel Computing*, vol. 39, no. 11, pp. 737–750, 2013.

[42] W. T. Tang, W. J. Tan, R. Ray, Y. W. Wong, W. Chen, S. Kuo, R. S. M. Goh, S. J. Turner, and W. Wong, “Accelerating sparse matrix-vector multiplication on GPUs using bit-representation-optimized schemes,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[43] J. W. Choi, A. Singh, and R. W. Vuduc, “Model-driven autotuning of sparse matrix-vector multiply on GPUs,” in *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2010, p. 115.

[44] S. Hong and H. Kim, “An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness,” *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, p. 152, 2009.

[45] B. He, N. K. Govindaraju, Q. Luo, and B. Smith, “Efficient gather and scatter operations on graphics processors,” in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, vol. 26, 2007, pp. 80–113.

[46] A. H. El Zein and A. P. Rendell, “From Sparse Matrix to Optimal GPU CUDA Sparse Matrix Vector Product Implementation,” in *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, 2010, pp. 808–813.

[47] P. Guo, L. Wang, and P. Chen, “A Performance Modeling and Optimization Analysis Tool for Sparse Matrix-Vector Multiplication on GPUs,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 5, pp. 1112–1123, 2014.

[48] R. Vuduc, A. Chandramowlishwaran, and J. Choi, “On the Limits of GPU Acceleration,” in *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, 2010, p. 13.

[49] J. D. Davis and E. S. Chung, “SpMV : A Memory-Bound Application on the GPU Stuck Between a Rock and a Hard Place,” *Microsoft Technical Report*, Rapport technique MSR-TR-2012-95, 2012.

ANNEXE A ALGORITHME SPMV POUR LE FORMAT COO

Les algorithmes effectuant la SpMV où la matrice creuse est stockée en format COO proviennent de la bibliothèque CUSP [14].

```
__global__ void cooFlatKernel(
        const int NNZflat,
        const int interval_size,
        const int * row,
        const int * col,
        const float * val,
        const float * x,
        float * y,
        int * temp_rows,
        float * temp_ results)
{
    __shared__ volatile int rows[48 *(Sb/32)];
    __shared__ volatile float results[Sb];

    // global thread index
    const int thread_id = Sb * blockIdx.x + threadIdx.x;
    // thread index within the warp
    const int thread_lane = threadIdx.x & (Sw - 1);
    // global warp index
    const int warp_id = thread_id / Sw;

    // warp's offset into row, col, val
    const int begin = warp_id * interval_size;
    // end of warps's work
    const int end = thrust::min(begin + interval_size, num_nonzeros);

    // thread's index into padded rows array
    const int idx = 16 * (threadIdx.x/32 + 1) + threadIdx.x;

    // fill padding with invalid row index
    rows[idx - 16] = -1;

    if(interval_begin >= interval_end)    // warp has no work to do
        return;

    if (thread_lane == 31)
    {
        // initialize the carry in values
        rows[idx] = row[interval_begin];
        vals[threadIdx.x] = float(0);
    }

    for(int n = begin + thread_lane; n < end; n += Sw)
    {
        int r = row[n];                   // row index (i)
        int c = col[n];
        float result = val[n] * x[c];     // A(i,j) * x(j)

        rows[idx]         = r;
```

```
        results[threadIdx.x] = result;

        if(r == rows[idx -  1])
        {
            result = result + results[threadIdx.x -  1];
            results[threadIdx.x] = result;
        }
        if(r == rows[idx -  2])
        {
            result = result + vals[threadIdx.x -  2];
            results[threadIdx.x] = result;
        }
        if(r == rows[idx -  4])
        {
            result = result + vals[threadIdx.x -  4];
            results[threadIdx.x] = result;
        }
        if(row == rows[idx -  8])
        {
            result = result + results[threadIdx.x -  8];
            results[threadIdx.x] = result;
        }
        if(row == rows[idx - 16])
        {
            result = result + results[threadIdx.x - 16];
            results[threadIdx.x] = result;
        }

        if(thread_lane < 31 && row != rows[idx + 1])
            y[row] += vals[threadIdx.x];    // row terminated
    }

    if(thread_lane == 31)
    {
        // write the carry out values
        temp_rows[warp_id] = rows[idx];
        temp_ results[warp_id] = results[threadIdx.x];
    }
}
```

Figure A.1 : Algorithme du noyau basé sur *spmv_coo_flat_kernel* utilisé dans le mémoire

```
__device__ void segreduce_block(const int * idx, float * val)
{
    float left = 0;

    if( threadIdx.x >= 1 && idx[threadIdx.x] == idx[threadIdx.x - 1] )
        left = val[threadIdx.x -   1];
    __syncthreads();
    val[threadIdx.x] += left;
    left = 0;
    __syncthreads();

    if( threadIdx.x >=   2 && idx[threadIdx.x] == idx[threadIdx.x -   2] )
        left = val[threadIdx.x -   2];
    __syncthreads();
    val[threadIdx.x] += left;
    left = 0;
    __syncthreads();

    if( threadIdx.x >=   4 && idx[threadIdx.x] == idx[threadIdx.x -   4] )
        left = val[threadIdx.x -   4];
    __syncthreads();
    val[threadIdx.x] += left;
    left = 0;
    __syncthreads();

    if( threadIdx.x >=   8 && idx[threadIdx.x] == idx[threadIdx.x -   8] )
        left = val[threadIdx.x -   8];
    __syncthreads();
    val[threadIdx.x] += left;
    left = 0;
    __syncthreads();

    if( threadIdx.x >=  16 && idx[threadIdx.x] == idx[threadIdx.x -  16] )
        left = val[threadIdx.x -  16];
    __syncthreads();
    val[threadIdx.x] += left;
    left = 0;
    __syncthreads();

    if( threadIdx.x >=  32 && idx[threadIdx.x] == idx[threadIdx.x -  32] )
        left = val[threadIdx.x -  32];
    __syncthreads();
    val[threadIdx.x] += left;
    left = 0;
    __syncthreads();

    if( threadIdx.x >=  64 && idx[threadIdx.x] == idx[threadIdx.x -  64] )
        left = val[threadIdx.x -  64];
    __syncthreads();
    val[threadIdx.x] += left;
    left = 0;
    __syncthreads();

    if( threadIdx.x >= 128 && idx[threadIdx.x] == idx[threadIdx.x - 128] )
        left = val[threadIdx.x - 128];
    __syncthreads();
    val[threadIdx.x] += left;
```

```
    left = 0;
    __syncthreads();

    if( threadIdx.x >= 256 && idx[threadIdx.x] == idx[threadIdx.x - 256] )
        left = val[threadIdx.x - 256];
    __syncthreads();
    val[threadIdx.x] += left;
    left = 0;
    __syncthreads();
}

__global__ void cooReduceUpdateKernel (
        const int Wflat,
        const int * temp_rows,
        const float * temp_results,
        float * y)
{
    __shared__ int rows[Sb + 1];
    __shared__ float results[Sb + 1];

    const int end = Wflat - (Wflat & (Sb - 1));

    if (threadIdx.x == 0)
    {
        rows[Sb] = (int) -1;
        results[Sb] = (float)  0;
    }
    __syncthreads();

    int i = threadIdx.x;

    while (i < end)
    {
        // do full blocks
        rows[threadIdx.x] = temp_rows[i];
        results[threadIdx.x] = temp_ results[i];

        __syncthreads();

        segreduce_block(rows, results);

        if (rows[threadIdx.x] != rows[threadIdx.x + 1])
            y[rows[threadIdx.x]] += results[threadIdx.x];

        __syncthreads();

        i += Sb;
    }

    if (end < Wflat){
        if (i < Wflat){
            rows[threadIdx.x] = temp_rows[i];
            results[threadIdx.x] = temp_ results[i];
        } else {
            rows[threadIdx.x] = (int) -1;
            results[threadIdx.x] = (float)  0;
        }
```

```
        __syncthreads();

        segreduce_block(rows, results);

        if (i < Wflat)
            if (rows[threadIdx.x] != rows[threadIdx.x + 1])
                y[rows[threadIdx.x]] += results[threadIdx.x];
    }
}
```

Figure A.2 : Algorithme du noyau basé sur *spmv_coo_reduce_update_kernel* utilisé dans le mémoire

ANNEXE B RÉSULTATS DU FORMAT CSR

Tableau B.1 : Nombre de transactions estimé et mesuré pour le format CSR avec un fil par ligne

#	Matrices	Transactions estimées			Transactions mesurées			D_T
		Lectures	Écri-tures	Total	Lectures	Écri-tures	Total	
1	Harvard500	3697	16	3713	3775	16	3791	2.058%
2	rail4284	33735771	134	33735905	33735589	134	33735723	-0.001%
3	wiki-Vote	279487	260	279747	280811	260	281071	0.471%
4	California	25144	302	25446	26023	302	26325	3.339%
5	wb-cs-stanford	49396	310	49706	50352	310	50662	1.887%
6	linverse	67839	375	68214	67841	375	68216	0.003%
7	ex11	2310687	520	2311207	2310655	520	2311175	-0.001%
8	OPF_6000	355632	935	356567	370017	935	370952	3.878%
9	pdb1HYS	9346580	1139	9347719	9346676	1139	9347815	0.001%
10	OPF_10000	566493	1372	567865	584747	1372	586119	3.114%
11	rma10	4907606	1464	4909070	4910036	1464	4911500	0.049%
12	conf5_4-8x8-05	3982720	1536	3984256	3982720	1536	3984256	0.000%
13	nasasrb	5588170	1715	5589885	5590030	1715	5591745	0.033%
14	cant	8451509	1952	8453461	8452292	1952	8454244	0.009%
15	finan512	610509	2336	612845	614991	2336	617327	0.726%
16	consph	12617835	2605	12620440	12617932	2605	12620537	0.001%
17	hcircuit	373936	3303	377239	376511	3303	379814	0.678%
18	cop20k_A	5321410	3788	5325198	5321847	3788	5325635	0.008%
19	shipsec1	6967080	4403	6971483	6975004	4403	6979407	0.114%
20	scircuit	924802	5344	930146	951051	5344	956395	2.745%
21	mac_econ_fwd500	1566480	6454	1572934	1624406	6454	1630860	3.552%
22	pwtk	23846553	6810	23853363	23847702	6810	23854512	0.005%
23	web-Stanford	5062729	8810	5071539	5123334	8810	5132144	1.181%
24	cnr-2000	5566969	10174	5577143	5592558	10174	5602732	0.457%
25	amazon0505	4785932	12820	4798752	4874471	12820	4887291	1.812%
26	neos	657883	14973	672856	675048	14973	690021	2.488%
27	mc2depi	792408	16433	808841	800206	16433	816639	0.955%
28	flickr	26825616	25653	26851269	26930545	25653	26956198	0.389%
29	web-Google	10346536	28639	10375175	10487440	28639	10516079	1.340%
30	webbase-1M	2876940	31251	2908191	2902794	31251	2934045	0.881%

Tableau B.2 : Nombre de requêtes estimé et mesuré pour le format CSR avec un fil par ligne de la matrice

#	Matrices	Requêtes estimées			Requêtes mesurées			D_R
		Lectures	Écri-tures	Total	Lectures	Écri-tures	Total	
1	Harvard500	1355	16	1371	1382	16	1398	1.931%
2	rail4284	4809034	134	4809168	4809244	134	4809378	0.004%
3	wiki-Vote	101881	260	102141	102259	260	102519	0.369%
4	California	11359	302	11661	11782	302	12084	3.500%
5	wb-cs-stanford	25247	310	25557	25679	310	25989	1.662%
6	linverse	10875	375	11250	10875	375	11250	0.000%
7	ex11	117314	520	117834	117788	520	118308	0.401%
8	OPF_6000	37615	935	38550	40246	935	41181	6.389%
9	pdb1HYS	514264	1139	515403	515875	1139	517014	0.312%
10	OPF_10000	58625	1372	59997	62588	1372	63960	6.196%
11	rma10	338376	1464	339840	339582	1464	341046	0.354%
12	conf5_4-8x8-05	182784	1536	184320	182784	1536	184320	0.000%
13	nasasrb	284956	1715	286671	289417	1715	291132	1.532%
14	cant	430528	1952	432480	436186	1952	438138	1.291%
15	finan512	147808	2336	150144	147808	2336	150144	0.000%
16	consph	610823	2605	613428	611522	2605	614127	0.114%
17	hcircuit	91788	3303	95091	92238	3303	95541	0.471%
18	cop20k_A	300451	3788	304239	301549	3788	305337	0.360%
19	shipsec1	565312	4403	569715	572134	4403	576537	1.183%
20	scircuit	236654	5344	241998	242942	5344	248286	2.533%
21	mac_econ_fwd500	447125	6454	453579	466016	6454	472470	3.998%
22	pwtk	1121682	6810	1128492	1123728	6810	1130538	0.181%
23	web-Stanford	1265875	8810	1274685	1280824	8810	1289634	1.159%
24	cnr-2000	982859	10174	993033	995198	10174	1005372	1.227%
25	amazon0505	410240	12820	423060	446900	12820	459720	7.974%
26	neos	192972	14973	207945	201384	14973	216357	3.888%
27	mc2depi	230056	16433	246489	237718	16433	254151	3.015%
28	flickr	8640738	25653	8666391	8676591	25653	8702244	0.412%
29	web-Google	2075075	28639	2103714	2118875	28639	2147514	2.040%
30	webbase-1M	983946	31251	1015197	1004544	31251	1035795	1.989%

Tableau B.3 : Nombre de transactions estimé et mesuré pour le format CSR avec une chaîne par ligne

#	Matrices	Transactions estimées			Transactions mesurées			D_T
		Lectures	Écri-tures	Total	Lectures	Écri-tures	Total	
1	Harvard500	3197	500	3697	3208	500	3708	0.30%
2	rail4284	3756472	4284	3760756	3761752	4284	3766036	0.14%
3	wiki-Vote	110271	8297	118568	111373	8297	119670	0.92%
4	California	42423	9664	52087	42458	9664	52122	0.07%
5	wb-cs-stanford	51514	9914	61428	51652	9914	61566	0.22%
6	linverse	68611	11999	80610	68611	11999	80610	0.00%
7	ex11	361426	16614	378040	377091	16614	393705	3.98%
8	OPF_6000	234233	29902	264135	234659	29902	264561	0.16%
9	pdb1HYS	1075231	36417	1111648	1119406	36417	1155823	3.82%
10	OPF_10000	349368	43887	393255	350127	43887	394014	0.19%
11	rma10	721666	46835	768501	764892	46835	811727	5.33%
12	conf5_4-8x8-05	852480	49152	901632	995328	49152	1044480	13.68%
13	nasasrb	839932	54870	894802	981332	54870	1036202	13.65%
14	cant	1292052	62451	1354503	1360039	62451	1422490	4.78%
15	finan512	658305	74752	733057	660769	74752	735521	0.34%
16	consph	2033939	8334	2042273	2063965	83334	2147299	4.89%
17	hcircuit	700327	105676	806003	700566	105676	806242	0.03%
18	cop20k_A	1705404	121192	1826596	1744479	121192	1865671	2.09%
19	shipsec1	1987453	140874	2128327	2065667	140874	2206541	3.54%
20	scircuit	1222781	170998	1393779	1224011	170998	1395009	0.09%
21	mac_econ_fwd500	1613062	206500	1819562	1614382	206500	1820882	0.07%
22	pwtk	3260845	217918	3478763	3892079	217918	4109997	15.36%
23	web-Stanford	3594871	281903	3876774	3616512	281903	3898415	0.56%
24	cnr-2000	2212112	325557	2537669	2297022	325557	2622579	3.24%
25	amazon0505	4096622	410236	4506858	4096622	410236	4506858	0.00%
26	neos	3400335	479119	3879454	3400335	479119	3879454	0.00%
27	mc2depi	3794177	525825	4320002	3794177	525825	4320002	0.00%
28	flickr	11186287	820878	12007165	11241279	820878	12062157	0.46%
29	web-Google	8696311	916428	9612739	8701234	916428	9617662	0.05%
30	webbase-1M	6562897	1000005	7562902	6573755	1000005	7573760	0.14%

Tableau B.4 : Nombre de requêtes estimé et mesuré pour le format CSR avec une chaîne par ligne

#	Matrices	Requêtes estimées			Requêtes mesurées			D_R
		Lectures	Écri-tures	Total	Lectures	Écri-tures	Total	
1	Harvard500	2530	500	3030	2542	500	3042	0.39%
2	rail4284	1072761	4284	1077045	1078938	4284	1083222	0.57%
3	wiki-Vote	41077	8297	49374	42649	8297	50946	3.09%
4	California	34460	9664	44124	34499	9664	44163	0.09%
5	wb-cs-stanford	41566	9914	51480	41725	9914	51639	0.31%
6	linverse	59995	11999	71994	59995	11999	71994	0.00%
7	ex11	162948	16614	179562	179421	16614	196035	8.40%
8	OPF_6000	150167	29902	180069	150788	29902	180690	0.34%
9	pdb1HYS	533157	36417	569574	580398	36417	616815	7.66%
10	OPF_10000	220530	43887	264417	221625	43887	265512	0.41%
11	rma10	368347	46835	415182	428887	46835	475722	12.73%
12	conf5_4-8x8-05	393216	49152	442368	540672	49152	589824	25.00%
13	nasasrb	429255	54870	484125	581202	54870	636072	23.89%
14	cant	610011	62451	672462	683739	62451	746190	9.88%
15	finan512	376832	74752	451584	379904	74752	454656	0.68%
16	consph	860628	83334	943962	892098	83334	975432	3.23%
17	hcircuit	529976	105676	635652	530312	105676	635988	0.05%
18	cop20k_A	597890	121192	719082	652181	121192	773373	7.02%
19	shipsec1	797919	140874	938793	891102	140874	1031976	9.03%
20	scircuit	857144	170998	1028142	858680	170998	1029678	0.15%
21	mac_econ_fwd500	1034000	206500	1240500	1035500	206500	1242000	0.12%
22	pwtk	1736423	217918	1954341	2372654	217918	2590572	24.56%
23	web-Stanford	1454638	281903	1736541	1490317	281903	1772220	2.01%
24	cnr-2000	1520244	325557	1845801	1625802	325557	1951359	5.41%
25	amazon0505	2010881	410236	2421117	2010881	410236	2421117	0.00%
26	neos	2395595	479119	2874714	2395595	479119	2874714	0.00%
27	mc2depi	2629125	525825	3154950	2629125	525825	3154950	0.00%
28	flickr	3961779	820878	4782657	4047441	820878	4868319	1.76%
29	web-Google	4063974	916428	4980402	4072596	916428	4989024	0.17%
30	webbase-1M	5054820	1000005	6054825	5067558	1000005	6067563	0.21%

ANNEXE C RÉSULTATS DU FORMAT ELL

Tableau C.1 : Nombre de transactions estimé et mesuré pour le format ELLPACK

#	Matrices	Transactions estimées			Transactions mesurées			D_T
		Lectures	Écri-tures	Total	Lectures	Écri-tures	Total	
1	Harvard500	7140	16	7156	7140	16	7156	0.00%
2	rail4284	27531491	134	27531625	27531316	134	27531450	0.00%
3	wiki-Vote	594320	260	594580	594320	260	594580	0.00%
4	California	63799	302	64101	63799	302	64101	0.00%
5	wb-cs-stanford	189274	310	189584	189274	310	189584	0.00%
6	linverse	19491	375	19866	19491	375	19866	0.00%
7	ex11	281820	520	282340	281820	520	282340	0.00%
8	OPF_6000	250835	935	251770	250835	935	251770	0.00%
9	pdb1HYS	1424454	1139	1425593	1424454	1139	1425593	0.00%
10	OPF_10000	334088	1372	335460	334088	1372	335460	0.00%
11	rma10	996984	1464	998448	996984	1464	998448	0.00%
12	conf5_4-8x8-05	264064	1536	265600	264064	1536	265600	0.00%
13	nasasrb	1353917	1715	1355632	1353917	1715	1355632	0.00%
14	cant	1001783	1952	1003735	1001783	1952	1003735	0.00%
15	finan512	283565	2336	285901	283565	2336	285901	0.00%
16	consph	1389620	2605	1392225	1389620	2605	1392225	0.00%
17	hcircuit	8792367	3303	8795670	8792367	3303	8795670	0.00%
18	cop20k_A	1793309	3788	1797097	1793309	3788	1797097	0.00%
19	shipsec1	1864025	4403	1868428	1864025	4403	1868428	0.00%
20	scircuit	4051871	5344	4057215	4051871	5344	4057215	0.00%
21	mac_econ_fwd500	1250298	6454	1256752	1250298	6454	1256752	0.00%
22	pwtk	3885629	6810	3892439	3885629	6810	3892439	0.00%
23	web-Stanford	7173938	8810	7182748	7173938	8810	7182748	0.00%
24	cnr-2000	-	-	-	-	-	-	-
25	amazon0505	2947792	12820	2960612	2947792	12820	2960612	0.00%
26	neos	1136074	14973	1151047	1136074	14973	1151047	0.00%
27	mc2depi	333257	16433	349690	333256	16433	349689	0.00%
28	flickr	-	-	-	-	-	-	-
29	web-Google	30584332	28639	30612971	30584332	28639	30612971	0.00%
30	webbase-1M	-	-	-	-	-	-	-

Tableau C.2 : Nombre de requêtes estimé et mesuré pour le format ELLPACK

#	Matrices	Requêtes estimées			Requêtes mesurées			D_R
		Lectures	Écri-tures	Total	Lectures	Écri-tures	Total	
1	Harvard500	4002	16	4018	9360	16	9376	57,15%
2	rail4284	10734232	134	10734366	22585164	134	22585298	52,47%
3	wiki-Vote	299754	260	300014	696540	260	696800	56,94%
4	California	56698	302	57000	148584	302	148886	61,72%
5	wb-cs-stanford	102288	310	102598	257610	310	257920	60,22%
6	linverse	10125	375	10500	10125	375	10500	0,00%
7	ex11	124316	520	124836	140400	520	140920	11,41%
8	OPF_6000	94890	935	95825	213180	935	214115	55,25%
9	pdb1HYS	573680	1139	574819	697068	1139	698207	17,67%
10	OPF_10000	116830	1372	118202	238728	1372	240100	50,77%
11	rma10	435912	1464	437376	636840	1464	638304	31,48%
12	conf5_4-8x8-05	179712	1536	181248	179712	1536	181248	0,00%
13	nasasrb	661024	1715	662739	1420020	1715	1421735	53,39%
14	cant	436672	1952	438624	456768	1952	458720	4,38%
15	finan512	223904	2336	226240	385440	2336	387776	41,66%
16	consph	614747	2605	617352	633015	2605	635620	2,87%
17	hcircuit	4677685	3303	4680988	13862691	3303	13865994	66,24%
18	cop20k_A	502078	3788	505866	920484	3788	924272	45,27%
19	shipsec1	670408	4403	674811	898212	4403	902615	25,24%
20	scircuit	2037076	5344	2042420	5659296	5344	5664640	63,94%
21	mac_econ_fwd500	573454	6454	579908	851928	6454	858382	32,44%
22	pwtk	1964508	6810	1971318	3677400	6810	3684210	46,49%
23	web-Stanford	3078720	8810	3087530	6739650	8810	6748460	54,25%
24	cnr-2000	-	-	-	-	-	-	-
25	amazon0505	384600	12820	397420	384600	12820	397420	0,00%
26	neos	542901	14973	557874	1302651	14973	1317624	57,66%
27	mc2depi	197192	16433	213625	197196	16433	213629	0,00%
28	flickr	-	-	-	-	-	-	-
29	web-Google	14404582	28639	14433221	39178152	28639	39206791	63,19%
30	webbase-1M	-	-	-	-	-	-	-

Tableau C.3 : Nombre de requêtes estimé et mesuré de chaque vecteur de représentation du format ELL

	Matrice	*Rd*		*Ri* ou *Rx*		*Ry*	
		Modèle	Profileur	Modèle	Profileur	Modèle	Profileur
1	Harvard500	3120	3120	441	3120	16	16
2	rail4284	7528388	7528388	1602922	7528388	134	134
3	wiki-Vote	232180	232180	33787	232180	260	260
4	California	49528	49528	3585	49528	302	302
5	wb-cs-stanford	85870	85870	8209	85870	310	310
6	linverse	3375	3375	3375	3375	375	375
7	ex11	46800	46800	38758	46800	520	520
8	OPF_6000	71060	71060	11915	71060	935	935
9	pdb1HYS	232356	232356	170662	232356	1139	1139
10	OPF_10000	79576	79576	18627	79576	1372	1372
11	rma10	212280	212280	111816	212280	1464	1464
12	conf5_4-8x8-05	59904	59904	59904	59904	1536	1536
13	nasasrb	473340	473340	93842	473340	1715	1715
14	cant	152256	152256	142208	152256	1952	1952
15	finan512	128480	128480	47712	128480	2336	2336
16	consph	211005	211005	201871	211005	2605	2605
17	hcircuit	4620897	4620897	28394	4620897	3303	3303
18	cop20k_A	306828	306828	97625	306828	3788	3788
19	shipsec1	299404	299404	185502	299404	4403	4403
20	scircuit	1886432	1886432	75322	1886432	5344	5344
21	mac_econ_fwd500	283976	283976	144739	283976	6454	6454
22	pwtk	1225800	1225800	369354	1225800	6810	6810
23	web-Stanford	2246550	2246550	416085	2246550	8810	8810
24	cnr-2000	-	-	-	-	-	-
25	amazon0505	128200	128200	128200	128200	12820	12820
26	neos	434217	434217	54342	434217	14973	14973
27	mc2depi	65732	65732	65730	65732	16433	16433
28	flickr	-	-	-	-	-	-
29	web-Google	13059384	13059384	672599	13059384	28639	28639
30	webbase-1M	-	-	-	-	-	-

ANNEXE D RÉSULTATS DU FORMAT COO

Tableau D.1 : Nombre de transactions total estimé et mesuré pour le format COO

#	Matrices	Transactions estimées			Transactions mesurées			D_T
		Lectures	Écri-tures	Total	Lectures	Écri-tures	Total	
1	Harvard500	889	278	1167	891	280	1171	0.34%
2	rail4284	3796539	710671	4507210	3799219	713350	4512569	0.12%
3	wiki-Vote	81565	8349	89914	81837	8621	90458	0.60%
4	California	10177	2022	12199	10214	2059	12273	0.60%
5	wb-cs-stanford	12368	3919	16287	12328	3879	16207	-0.49%
6	linverse	19361	9392	28753	19443	9474	28917	0.57%
7	ex11	336181	85578	421759	336802	86199	423001	0.29%
8	OPF_6000	92723	28768	121491	91438	27483	118921	-2.16%
9	pdb1HYS	1034966	308438	1343404	1038628	312100	1350728	0.54%
10	OPF_10000	141703	44157	185860	139999	42453	182452	-1.87%
11	rma10	642153	190112	832265	643558	191517	835075	0.34%
12	conf5_4-8x8-05	736007	169063	905070	737728	170784	908512	0.38%
13	nasasrb	772400	221748	994148	774683	224031	998714	0.46%
14	cant	1187351	313693	1501044	1190150	316492	1506642	0.37%
15	finan512	212364	59260	271624	212875	59771	272646	0.37%
16	consph	1908913	456146	2365059	1914264	461497	2375761	0.45%
17	hcircuit	152769	51920	204689	154908	54059	208967	2.05%
18	cop20k_A	1319923	239789	1559712	1323583	243449	1567032	0.47%
19	shipsec1	1443401	331995	1775396	1445053	333647	1778700	0.19%
20	scircuit	375980	99534	475514	376342	99896	476238	0.15%
21	mac_econ_fwd500	583062	131252	714314	583553	131743	715296	0.14%
22	pwtk	3041517	942959	3984476	3052197	953639	4005836	0.53%
23	web-Stanford	2615834	225796	2841630	2617854	227816	2845670	0.14%
24	cnr-2000	1014107	304246	1318353	1017341	307480	1324821	0.49%
25	amazon0505	2379765	334395	2714160	2384794	339424	2724218	0.37%
26	neos	615304	172313	787617	615276	172285	787561	-0.01%
27	mc2depi	604277	227658	831935	605327	228708	834035	0.25%
28	flickr	8464645	750904	9215549	8490842	777101	9267943	0.57%
29	web-Google	5951792	528112	6479904	5954581	530900	6485481	0.09%
30	webbase-1M	1060709	305518	1366227	1092029	336838	1428867	4.38%

Tableau D.2 : Nombre de requêtes total estimé et mesuré pour le format COO

#	Matrices	Requêtes estimées			Requêtes mesurées			D_R
		Lectures	Écri-tures	Total	Lectures	Écri-tures	Total	
1	Harvard500	484	238	558	484	238	722	0.00%
2	rail4284	1767170	709292	1771210	1767169	709291	2476460	-0.87%
3	wiki-Vote	17804	8084	19408	17804	8084	25888	-0.74%
4	California	3015	1503	3510	3015	1503	4518	-0.38%
5	wb-cs-stanford	6760	3307	7765	6760	3307	10067	-0.64%
6	linverse	17994	8997	20993	17994	8997	26991	-0.65%
7	ex11	187458	84621	203521	187458	84621	272079	-0.77%
8	OPF_6000	51410	25658	59900	51410	25658	77068	-0.68%
9	pdb1HYS	714135	306816	749405	714134	306815	1020949	-0.82%
10	OPF_10000	79953	39933	93206	79953	39933	119886	-0.69%
11	rma10	406078	187726	448236	406078	187726	593804	-0.75%
12	conf5_4-8x8-05	347137	167425	394754	347136	167424	514560	-0.72%
13	nasasrb	470755	219757	523180	470755	219757	690512	-0.74%
14	cant	686417	310727	686417	686417	310727	997144	-0.77%
15	finan512	111232	55264	111232	111232	55264	166496	-0.68%
16	consph	1016435	452954	1016435	1016435	452954	1469389	-0.79%
17	hcircuit	95759	47660	95759	95759	47660	143419	-0.68%
18	cop20k_A	483153	237123	483153	483153	237123	720276	-0.70%
19	shipsec1	659383	324868	659383	659383	324868	984251	-0.70%
20	scircuit	179461	89563	179461	179461	89563	269024	-0.69%
21	mac_econ_fwd500	238567	119188	238567	238567	119188	357755	-0.68%
22	pwtk	2016299	935885	2016299	2016298	935884	2952182	-0.75%
23	web-Stanford	427315	210520	427315	427315	210520	637835	-0.70%
24	cnr-2000	603307	295417	898724	600170	292280	892450	-0.70%
25	amazon0505	642630	321276	963906	639357	318003	957360	-0.68%
26	neos	292284	146126	438410	290795	144637	435432	-0.68%
27	mc2depi	401999	200997	602996	399950	198948	598898	-0.68%
28	flickr	1688262	746692	2434954	1678661	737091	2415752	-0.79%
29	web-Google	975015	486387	1461402	970030	481402	1451432	-0.69%
30	webbase-1M	579134	281924	861058	576101	278891	854992	-0.71%

Tableau D.3 : Nombre de transactions estimé et mesuré pour la section SpMV parallèle du format COO

#	Matrices	Transactions estimées			Transactions mesurées			D_T
		Lectures	Écri-tures	Total	Lectures	Écri-tures	Total	
1	Harvard500	807	250	1057	807	250	1057	0.00%
2	rail4284	3773121	709293	4482414	3773121	709292	4482413	0.00%
3	wiki-Vote	81241	8265	89506	81241	8265	89506	0.00%
4	California	9798	1763	11561	9798	1763	11561	0.00%
5	wb-cs-stanford	11838	3549	15387	11838	3549	15387	0.00%
6	linverse	18742	8997	27739	18742	8997	27739	0.00%
7	ex11	333000	84621	417621	333000	84621	417621	0.00%
8	OPF_6000	89681	26300	115981	89681	26300	115981	0.00%
9	pdb1HYS	1024743	306817	1331560	1024742	306816	1331558	0.00%
10	OPF_10000	137333	40693	178026	137333	40693	178026	0.00%
11	rma10	635308	187833	823141	635308	187833	823141	0.00%
12	conf5_4-8x8-05	730625	167425	898050	730624	167424	898048	0.00%
13	nasasrb	765150	219776	984926	765150	219776	984926	0.00%
14	cant	1176472	310734	1487206	1176472	310734	1487206	0.00%
15	finan512	208866	56928	265794	208866	56928	265794	0.00%
16	consph	1894158	453195	2347353	1894158	453195	2347353	0.00%
17	hcircuit	150123	50342	200465	150123	50342	200465	0.00%
18	cop20k_A	1312820	237856	1550676	1312820	237856	1550676	0.00%
19	shipsec1	1430252	325880	1756132	1430252	325880	1756132	0.00%
20	scircuit	368249	93773	462022	368249	93773	462022	0.00%
21	mac_econ_fwd500	573513	124243	697756	573513	124243	697756	0.00%
22	pwtk	3011893	935909	3947802	3011892	935908	3947800	0.00%
23	web-Stanford	2602480	217028	2819508	2602480	217028	2819508	0.00%
24	cnr-2000	998667	295184	1293851	998667	295184	1293851	0.00%
25	amazon0505	2362433	323717	2686150	2362433	323717	2686150	0.00%
26	neos	596244	156275	752519	596244	156275	752519	0.00%
27	mc2depi	583249	210736	793985	583249	210736	793985	0.00%
28	flickr	8441678	747271	9188949	8441678	747271	9188949	0.00%
29	web-Google	5911834	498186	6410020	5911835	498186	6410021	0.00%
30	webbase-1M	1052747	303622	1356369	1052747	303622	1356369	0.00%

Tableau D.4 : Nombre de requêtes estimé et mesuré pour la section SpMV parallèle du format COO

#	Matrices	Requêtes estimées			Requêtes mesurées			D_R
		Lectures	Écri-tures	Total	Lectures	Écri-tures	Total	
1	Harvard500	18	9	27	18	9	27	0.00%
2	rail4284	24	12	36	24	12	36	0.00%
3	wiki-Vote	36	18	54	36	18	54	0.00%
4	California	54	27	81	54	27	81	0.00%
5	wb-cs-stanford	18	9	27	18	9	27	0.00%
6	linverse	18	9	27	18	9	27	0.00%
7	ex11	84	42	126	84	42	126	0.00%
8	OPF_6000	1502	749	2251	1502	749	2251	0.00%
9	pdb1HYS	484	238	722	484	238	722	0.00%
10	OPF_10000	1767170	709292	2476462	1767169	709291	2476460	0.00%
11	rma10	17804	8084	25888	17804	8084	25888	0.00%
12	conf5_4-8x8-05	3015	1503	4518	3015	1503	4518	0.00%
13	nasasrb	6760	3307	10067	6760	3307	10067	0.00%
14	cant	17994	8997	26991	17994	8997	26991	0.00%
15	finan512	187458	84621	272079	187458	84621	272079	0.00%
16	consph	51410	25658	77068	51410	25658	77068	0.00%
17	hcircuit	714135	306816	1020951	714134	306815	1020949	0.00%
18	cop20k_A	79953	39933	119886	79953	39933	119886	0.00%
19	shipsec1	406078	187726	593804	406078	187726	593804	0.00%
20	scircuit	347137	167425	514562	347136	167424	514560	0.00%
21	mac_econ_fwd500	470755	219757	690512	470755	219757	690512	0.00%
22	pwtk	686417	310727	997144	686417	310727	997144	0.00%
23	web-Stanford	111232	55264	166496	111232	55264	166496	0.00%
24	cnr-2000	1016435	452954	1469389	1016435	452954	1469389	0.00%
25	amazon0505	95759	47660	143419	95759	47660	143419	0.00%
26	neos	483153	237123	720276	483153	237123	720276	0.00%
27	mc2depi	659383	324868	984251	659383	324868	984251	0.00%
28	flickr	179461	89563	269024	179461	89563	269024	0.00%
29	web-Google	238567	119188	357755	238567	119188	357755	0.00%
30	webbase-1M	2016299	935885	2952184	2016298	935884	2952182	0.00%

Tableau D.5 : Nombre de transactions estimé et mesuré pour l'addition des résultats temporaires du format COO

#	Matrices	Transactions estimées			Transactions mesurées			D_T
		Lectures	Écri-tures	Total	Lectures	Écri-tures	Total	
1	Harvard500	22	16	38	24	18	42	16.67%
2	rail4284	23418	1378	24796	26098	4058	30156	20.54%
3	wiki-Vote	279	75	354	551	347	898	98.73%
4	California	269	237	506	306	274	580	24.18%
5	wb-cs-stanford	420	348	768	380	308	688	-21.05%
6	linverse	574	386	960	656	468	1124	25.00%
7	ex11	3081	937	4018	3702	1558	5260	33.55%
8	OPF_6000	2997	2459	5456	1712	1174	2886	-150.12%
9	pdb1HYS	10078	1592	11670	13741	5255	18996	53.31%
10	OPF_10000	4280	3446	7726	2576	1742	4318	-132.30%
11	rma10	6825	2275	9100	8230	3680	11910	34.14%
12	conf5_4-8x8-05	5382	1638	7020	7104	3360	10464	48.48%
13	nasasrb	7190	1960	9150	9473	4243	13716	48.20%
14	cant	10764	2936	13700	13563	5735	19298	41.27%
15	finan512	3498	2332	5830	4009	2843	6852	25.49%
16	consph	14675	2935	17610	20026	8286	28312	53.44%
17	hcircuit	2566	1562	4128	4705	3701	8406	90.92%
18	cop20k_A	7048	1922	8970	10708	5582	16290	68.36%
19	shipsec1	13069	6099	19168	14721	7751	22472	22.44%
20	scircuit	7611	5737	13348	7973	6099	14072	9.08%
21	mac_econ_fwd500	9484	6996	16480	9975	7487	17462	9.84%
22	pwtk	29544	7034	36578	40225	17715	57940	53.11%
23	web-Stanford	13269	8751	22020	15289	10771	26060	26.42%
24	cnr-2000	15320	9038	24358	18554	12272	30826	34.86%
25	amazon0505	17212	10654	27866	22241	15683	37924	45.22%
26	neos	19010	16028	35038	18982	16000	34982	-0.30%
27	mc2depi	21023	16921	37944	22073	17971	40044	9.51%
28	flickr	22817	3603	26420	49014	29800	78814	106.90%
29	web-Google	39883	29911	69794	42671	32699	75370	13.07%
30	webbase-1M	7962	1896	9858	39282	33216	72498	159.46%

Tableau D.6 : Nombre de requêtes estimé et mesuré pour l'addition des résultats temporaires du format COO

#	Matrices	Requêtes estimées			Requêtes mesurées			D_R
		Lectures	Écri-tures	Total	Lectures	Écri-tures	Total	
1	Harvard500	9	3	12	9	3	12	0.00%
2	rail4284	44077	22037	66114	33058	11018	44076	-66.66%
3	wiki-Vote	403	199	602	306	102	408	-63.40%
4	California	57	25	82	48	16	64	-37.50%
5	wb-cs-stanford	141	69	210	108	36	144	-61.11%
6	linverse	371	183	554	282	94	376	-63.12%
7	ex11	4281	2137	6418	3216	1072	4288	-66.23%
8	OPF_6000	1072	534	1606	807	269	1076	-65.68%
9	pdb1HYS	16970	8484	25454	12729	4243	16972	-66.64%
10	OPF_10000	1668	834	2502	1251	417	1668	-66.67%
11	rma10	9098	4548	13646	6825	2275	9100	-66.61%
12	conf5_4-8x8-05	7489	3745	11234	5616	1872	7488	-66.70%
13	nasasrb	10454	5224	15678	7845	2615	10460	-66.51%
14	cant	15655	7827	23482	11742	3914	15656	-66.65%
15	finan512	2326	1160	3486	1749	583	2332	-65.98%
16	consph	23475	11735	35210	17610	5870	23480	-66.61%
17	hcircuit	2003	999	3002	1506	502	2008	-66.00%
18	cop20k_A	10250	5124	15374	7689	2563	10252	-66.61%
19	shipsec1	13936	6966	20902	10455	3485	13940	-66.59%
20	scircuit	3748	1874	5622	2811	937	3748	-66.67%
21	mac_econ_fwd500	4973	2485	7458	3732	1244	4976	-66.51%
22	pwtk	45014	22504	67518	33765	11255	45020	-66.63%
23	web-Stanford	9034	4516	13550	6777	2259	9036	-66.61%
24	cnr-2000	12560	6278	18838	9423	3141	12564	-66.58%
25	amazon0505	13110	6552	19662	9837	3279	13116	-66.54%
26	neos	5962	2980	8942	4473	1491	5964	-66.58%
27	mc2depi	8202	4100	12302	6153	2051	8204	-66.60%
28	flickr	38422	19208	57630	28821	9607	38428	-66.63%
29	web-Google	19943	9971	29914	14958	4986	19944	-66.65%
30	webbase-1M	12132	6066	18198	9099	3033	12132	-66.67%

Tableau D.7 : Nombre de transactions et de requêtes estimé et mesuré pour la section SpMV séquentielle du format COO

#	Matrices	Transactions estimées			Transactions mesurées			D_T et D_R
		Lectures	Écri-tures	Total	Lectures	Écri-tures	Total	
1	Harvard500	35	7	42	35	7	42	0.00%
2	rail4284	25	5	30	25	5	30	0.00%
3	wiki-Vote	70	14	84	70	14	84	0.00%
4	California	105	21	126	105	21	126	0.00%
5	wb-cs-stanford	130	26	156	130	26	156	0.00%
6	linverse	-	-	-	-	-	-	-
7	ex11	50	10	60	50	10	60	0.00%
8	OPF_6000	-	-	-	-	-	-	-
9	pdb1HYS	60	12	72	60	12	72	0.00%
10	OPF_10000	-	-	-	-	-	-	-
11	rma10	45	9	54	45	9	54	0.00%
12	conf5_4-8x8-05	110	22	132	110	22	132	0.00%
13	nasasrb	110	22	132	110	22	132	0.00%
14	cant	45	9	54	45	9	54	0.00%
15	finan512	100	20	120	100	20	120	0.00%
16	consph	45	9	54	45	9	54	0.00%
17	hcircuit	145	29	174	145	29	174	0.00%
18	cop20k_A	90	18	108	90	18	108	0.00%
19	shipsec1	20	4	24	20	4	24	0.00%
20	scircuit	-	-	-	-	-	-	-
21	mac_econ_fwd500	60	12	72	60	12	72	0.00%
22	pwtk	115	23	138	115	23	138	0.00%
23	web-Stanford	-	-	-	-	-	-	-
24	cnr-2000	80	16	96	80	16	96	0.00%
25	amazon0505	80	16	96	80	16	96	0.00%
26	neos	55	11	66	55	11	66	0.00%
27	mc2depi	80	16	96	80	16	96	0.00%
28	flickr	120	24	144	120	24	144	0.00%
29	web-Google	65	13	78	65	13	78	0.00%
30	webbase-1M	80	16	96	80	16	96	0.00%

ANNEXE E RÉSULTATS DU FORMAT HYB

Tableau E.1 : Nombre de transactions estimé et mesuré pour le format HYB

#	Matrices	Transactions estimées			Transactions mesurées			D_T
		Lectures	Écri-tures	Total	Lectures	Écri-tures	Total	
1	Harvard500	856	197	1053	875	198	1073	1.86%
2	rail4284	10130	1790	11920	10185	1826	12011	0.76%
3	wiki-Vote	13732	2209	15941	13706	2160	15866	-0.47%
4	California	19491	375	19866	19491	375	19866	0.00%
5	wb-cs-stanford	81867	7449	89316	82114	7685	89799	0.54%
6	linverse	118540	8220	126760	117746	7398	125144	-1.29%
7	ex11	187751	11088	198839	186875	10198	197073	-0.90%
8	OPF_6000	130035	10092	140127	130220	10257	140477	0.25%
9	pdb1HYS	129102	18218	147320	129492	18608	148100	0.53%
10	OPF_10000	349194	32544	381738	348062	31399	379461	-0.60%
11	rma10	272702	1449	274151	272709	1449	274158	0.00%
12	conf5_4-8x8-05	357088	30981	388069	357320	31184	388504	0.11%
13	nasasrb	618459	40253	658712	618571	40340	658911	0.03%
14	cant	264064	1536	265600	264064	1536	265600	0.00%
15	finan512	333257	16433	349690	333257	16433	349690	0.00%
16	consph	615444	2858	618302	615482	2887	618369	0.01%
17	hcircuit	697747	40253	738000	696887	39387	736274	-0.23%
18	cop20k_A	2544729	124330	2669059	2545120	124705	2669825	0.03%
19	shipsec1	1218323	46715	1265038	1218816	47196	1266012	0.08%
20	scircuit	830285	137588	967873	838601	145893	984494	1.69%
21	mac_econ_fwd500	2947792	12820	2960612	2947792	12820	2960612	0.00%
22	pwtk	1405996	53715	1459711	1404595	52307	1456902	-0.19%
23	web-Stanford	989860	5162	995022	989869	5162	995031	0.00%
24	cnr-2000	1256703	17809	1274512	1256714	18725	1275439	0.07%
25	amazon0505	1018186	190885	1209071	1019769	192460	1212229	0.26%
26	neos	1389620	2605	1392225	1389620	2605	1392225	0.00%
27	mc2depi	6150922	240149	6391071	6150660	239859	6390519	-0.01%
28	flickr	2223797	11941	2235738	2223924	12057	2235981	0.01%
29	web-Google	8466285	695507	9161792	8487172	716371	9203543	0.45%
30	webbase-1M	8113046	386934	8499980	8114083	388046	8502129	0.03%

Tableau E.2 : Nombre de requêtes estimé et mesuré pour le format HYB

#	Matrices	Requêtes estimées			Requêtes mesurées			D_R
		Lectures	Écri-tures	Total	Lectures	Écri-tures	Total	
1	Harvard500	522	183	705	540	183	723	2.49%
2	rail4284	3098	1374	4472	3108	1365	4473	0.02%
3	wiki-Vote	6669	1778	8447	6845	1761	8606	1.85%
4	California	10125	375	10500	10125	375	10500	0.00%
5	wb-cs-stanford	18206	7409	25615	18136	7320	25456	-0.62%
6	linverse	36963	6572	43535	39366	6515	45881	5.11%
7	ex11	56934	8945	65879	60413	8872	69285	4.92%
8	OPF_6000	61932	9509	71441	62367	9444	71811	0.52%
9	pdb1HYS	74998	15048	90046	74869	14919	89788	-0.29%
10	OPF_10000	118427	25083	143510	119733	24882	144615	0.76%
11	rma10	119287	1002	120289	125787	1002	126789	5.13%
12	conf5_4-8x8-05	151982	23553	175535	151930	23472	175402	-0.08%
13	nasasrb	181549	29388	210937	181341	29155	210496	-0.21%
14	cant	179712	1536	181248	179712	1536	181248	0.00%
15	finan512	197192	16433	213625	197196	16433	213629	0.00%
16	consph	279645	2792	282437	280077	2783	282860	0.15%
17	hcircuit	310825	38825	349650	319860	38408	358268	2.41%
18	cop20k_A	368724	111733	480457	367650	110644	478294	-0.45%
19	shipsec1	317069	46782	363851	347294	46341	393635	7.57%
20	scircuit	385385	122735	508120	384314	121654	505968	-0.43%
21	mac_econ_fwd500	384600	12820	397420	384600	12820	397420	0.00%
22	pwtk	452286	46214	498500	453106	45789	498895	0.08%
23	web-Stanford	429898	3194	433092	441703	3185	444888	2.65%
24	cnr-2000	495262	17337	512599	504735	17508	522243	1.85%
25	amazon0505	578404	184529	762933	598763	182672	781435	2.37%
26	neos	614747	2605	617352	633015	2605	635620	2.87%
27	mc2depi	931479	192882	1124361	929825	191193	1121018	-0.30%
28	flickr	1107505	11416	1118921	1112496	11374	1123870	0.44%
29	web-Google	1675711	692943	2368654	1672016	684174	2356190	-0.53%
30	webbase-1M	1850053	398202	2248255	1848242	392191	2240433	-0.35%

Tableau E.3 : Nombre de transactions estimé et mesuré pour la section ELLPACK du format HYB

#	Matrices	Transactions estimées			Transactions mesurées			D_T
		Lectures	Écri-tures	Total	Lectures	Écri-tures	Total	
1	Harvard500	315	16	331	315	16	331	0.000%
2	rail4284	6078230	134	6078364	6078141	134	6078275	-0.001%
3	wiki-Vote	11090	260	11350	11090	260	11350	0.000%
4	California	2450	302	2752	2450	302	2752	0.000%
5	wb-cs-stanford	7786	310	8096	7786	310	8096	0.000%
6	linverse	19491	375	19866	19491	375	19866	0.000%
7	ex11	270904	520	271424	270904	520	271424	0.000%
8	OPF_6000	95889	935	96824	95889	935	96824	0.000%
9	pdb1HYS	1212324	1139	1213463	1212324	1139	1213463	0.000%
10	OPF_10000	156134	1372	157506	156134	1372	157506	0.000%
11	rma10	584521	1464	585985	584521	1464	585985	0.000%
12	conf5_4-8x8-05	264064	1536	265600	264064	1536	265600	0.000%
13	nasasrb	612800	1715	614515	612800	1715	614515	0.000%
14	cant	983750	1952	985702	983750	1952	985702	0.000%
15	finan512	83124	2336	85460	83124	2336	85460	0.000%
16	consph	1389620	2605	1392225	1389620	2605	1392225	0.000%
17	hcircuit	104313	3303	107616	104313	3303	107616	0.000%
18	cop20k_A	1013554	3788	1017342	1013554	3788	1017342	0.000%
19	shipsec1	1266703	4403	1271106	1266703	4403	1271106	0.000%
20	scircuit	241662	5344	247006	241662	5344	247006	0.000%
21	mac_econ_fwd500	495634	6454	502088	495634	6454	502088	0.000%
22	pwtk	2211990	6810	2218800	2211990	6810	2218800	0.000%
23	web-Stanford	1270359	8810	1279169	1270359	8810	1279169	0.000%
24	cnr-2000	477491	10174	487665	477491	10174	487665	0.000%
25	amazon0505	2947792	12820	2960612	2947792	12820	2960612	0.000%
26	neos	326238	14973	341211	326238	14973	341211	0.000%
27	mc2depi	333257	16433	349690	333257	16433	349690	0.000%
28	flickr	654178	25653	679831	654178	25653	679831	0.000%
29	web-Google	4100233	28639	4128872	4100233	28639	4128872	0.000%
30	webbase-1M	340761	31251	372012	340761	31251	372012	0.000%

Tableau E.4 : Nombre de requêtes estimé et mesuré pour la section ELLPACK du format HYB

#	Matrices	Requêtes estimées			Requêtes mesurées			D_R
		Lectures	Écri-tures	Total	Lectures	Écri-tures	Total	
1	Harvard500	144	16	160	144	16	160	0.00%
2	rail4284	862928	134	863062	867114	134	867248	0.48%
3	wiki-Vote	2332	260	2592	2340	260	2600	0.31%
4	California	906	302	1208	906	302	1208	0.00%
5	wb-cs-stanford	3550	310	3860	3720	310	4030	4.22%
6	linverse	10125	375	10500	10125	375	10500	0.00%
7	ex11	118306	520	118826	124800	520	125320	5.18%
8	OPF_6000	25618	935	26553	28050	935	28985	8.39%
9	pdb1HYS	461924	1139	463063	471546	1139	472685	2.04%
10	OPF_10000	41738	1372	43110	45276	1372	46648	7.58%
11	rma10	232114	1464	233578	241560	1464	243024	3.89%
12	conf5_4-8x8-05	179712	1536	181248	179712	1536	181248	0.00%
13	nasasrb	277398	1715	279113	277830	1715	279545	0.15%
14	cant	427394	1952	429346	439200	1952	441152	2.68%
15	finan512	49056	2336	51392	49056	2336	51392	0.00%
16	consph	614747	2605	617352	633015	2605	635620	2.87%
17	hcircuit	49065	3303	52368	49545	3303	52848	0.91%
18	cop20k_A	230718	3788	234506	261372	3788	265160	11.56%
19	shipsec1	368614	4403	373017	369852	4403	374255	0.33%
20	scircuit	78666	5344	84010	80160	5344	85504	1.75%
21	mac_econ_fwd500	135534	6454	141988	135534	6454	141988	0.00%
22	pwtk	1098198	6810	1105008	1103220	6810	1110030	0.45%
23	web-Stanford	158580	8810	167390	158580	8810	167390	0.00%
24	cnr-2000	221968	10174	232142	244176	10174	254350	8.73%
25	amazon0505	384600	12820	397420	384600	12820	397420	0.00%
26	neos	134757	14973	149730	134757	14973	149730	0.00%
27	mc2depi	197192	16433	213625	197196	16433	213629	0.00%
28	flickr	148866	25653	174519	153918	25653	179571	2.81%
29	web-Google	601411	28639	630050	601419	28639	630058	0.00%
30	webbase-1M	187506	31251	218757	187506	31251	218757	0.00%

Tableau E.5 : Nombre de requêtes estimé et mesuré de chaque vecteur de représentation de la section ELL-PACK du format HYB

	Matrice	*Rd*		*Ri* ou *Rx*		*Ry*	
		Modèle	Profileur	Modèle	Profileur	Modèle	Profileur
1	Harvard500	48	48	48	48	16	16
2	rail4284	289038	289038	286945	289038	134	134
3	wiki-Vote	780	780	776	780	260	260
4	California	302	302	302	302	302	302
5	wb-cs-stanford	1240	1240	1155	1240	310	310
6	linverse	3375	3375	3375	3375	375	375
7	ex11	41600	41600	38353	41600	520	520
8	OPF_6000	9350	9350	8134	9350	935	935
9	pdb1HYS	157182	157182	152371	157182	1139	1139
10	OPF_10000	15092	15092	13323	15092	1372	1372
11	rma10	80520	80520	75797	80520	1464	1464
12	conf5_4-8x8-05	59904	59904	59904	59904	1536	1536
13	nasasrb	92610	92610	92394	92610	1715	1715
14	cant	146400	146400	140497	146400	1952	1952
15	finan512	16352	16352	16352	16352	2336	2336
16	consph	211005	211005	201871	211005	2605	2605
17	hcircuit	16515	16515	16275	16515	3303	3303
18	cop20k_A	87124	87124	71797	87124	3788	3788
19	shipsec1	123284	123284	122665	123284	4403	4403
20	scircuit	26720	26720	25973	26720	5344	5344
21	mac_econ_fwd500	45178	45178	45178	45178	6454	6454
22	pwtk	367740	367740	365229	367740	6810	6810
23	web-Stanford	52860	52860	52860	52860	8810	8810
24	cnr-2000	81392	81392	70288	81392	10174	10174
25	amazon0505	128200	128200	128200	128200	12820	12820
26	neos	44919	44919	44919	44919	14973	14973
27	mc2depi	65732	65732	65730	65732	16433	16433
28	flickr	51306	51306	48780	51306	25653	25653
29	web-Google	200473	200473	200469	200473	28639	28639
30	webbase-1M	62502	62502	62502	62502	31251	31251

Tableau E.6 : Nombre de transactions estimé et mesuré pour la section COO du format HYB

#	Matrices	Transactions estimées			Transactions mesurées			D_T
		Lectures	Écri-tures	Total	Lectures	Écri-tures	Total	
1	Harvard500	541	181	722	560	182	742	2.70%
2	rail4284	2034816	386800	2421616	2035942	387912	2423854	0.09%
3	wiki-Vote	70777	7189	77966	71024	7425	78449	0.62%
4	California	7680	1488	9168	7735	1524	9259	0.98%
5	wb-cs-stanford	5946	1899	7845	5920	1850	7770	-0.97%
6	linverse	-	-	-	-	-	-	-
7	ex11	1798	929	2727	1805	929	2734	0.26%
8	OPF_6000	22651	7285	29936	21857	6463	28320	-5.71%
9	pdb1HYS	44379	16670	61049	44390	17586	61976	1.50%
10	OPF_10000	31617	9716	41333	30741	8826	39567	-4.46%
11	rma10	113226	38789	152015	112366	37923	150289	-1.15%
12	conf5_4-8x8-05	-	-	-	-	-	-	-
13	nasasrb	2644	1143	3787	2682	1172	3854	1.74%
14	cant	6110	3210	9320	6119	3210	9329	0.10%
15	finan512	45978	15882	61860	46368	16272	62640	1.25%
16	consph	-	-	-	-	-	-	-
17	hcircuit	25722	6789	32511	25907	6954	32861	1.07%
18	cop20k_A	204769	42927	247696	205262	43408	248670	0.39%
19	shipsec1	139293	49312	188605	137892	47904	185796	-1.51%
20	scircuit	107532	27200	134732	106400	26055	132455	-1.72%
21	mac_econ_fwd500	122825	33799	156624	122937	33886	156823	0.13%
22	pwtk	11807	5131	16938	11934	5247	17181	1.41%
23	web-Stanford	1274370	115520	1389890	1274761	115895	1390656	0.06%
24	cnr-2000	540695	180711	721406	542278	182286	724564	0.44%
25	amazon0505	-	-	-	-	-	-	-
26	neos	30850	16008	46858	31082	16211	47293	0.92%
27	mc2depi	-	-	-	-	-	-	-
28	flickr	7812107	669854	8481961	7832994	690718	8523712	0.49%
29	web-Google	2050689	211510	2262199	2050427	211220	2261647	-0.02%
30	webbase-1M	489524	106337	595861	497840	114642	612482	2.71%

Tableau E.7 : Nombre de requêtes estimé et mesuré pour la section COO du format HYB

#	Matrices	Requêtes estimées			Requêtes mesurées			D_R
		Lectures	Écri-tures	Total	Lectures	Écri-tures	Total	
1	Harvard500	541	181	722	560	182	742	2.70%
2	rail4284	2034816	386800	2421616	2035942	387912	2423854	0.09%
3	wiki-Vote	70777	7189	77966	71024	7425	78449	0.62%
4	California	7680	1488	9168	7735	1524	9259	0.98%
5	wb-cs-stanford	5946	1899	7845	5920	1850	7770	-0.97%
6	Linverse	-	-	-	-	-	-	-
7	ex11	1798	929	2727	1805	929	2734	0.26%
8	OPF_6000	22651	7285	29936	21857	6463	28320	-5.71%
9	pdb1HYS	44379	16670	61049	44390	17586	61976	1.50%
10	OPF_10000	31617	9716	41333	30741	8826	39567	-4.46%
11	rma10	113226	38789	152015	112366	37923	150289	-1.15%
12	conf5_4-8x8-05	-	-	-	-	-	-	-
13	nasasrb	2644	1143	3787	2682	1172	3854	1.74%
14	cant	6110	3210	9320	6119	3210	9329	0.10%
15	finan512	45978	15882	61860	46368	16272	62640	1.25%
16	consph	-	-	-	-	-	-	-
17	hcircuit	25722	6789	32511	25907	6954	32861	1.07%
18	cop20k_A	204769	42927	247696	205262	43408	248670	0.39%
19	shipsec1	139293	49312	188605	137892	47904	185796	-1.51%
20	scircuit	107532	27200	134732	106400	26055	132455	-1.72%
21	mac_econ_fwd500	122825	33799	156624	122937	33886	156823	0.13%
22	pwtk	11807	5131	16938	11934	5247	17181	1.41%
23	web-Stanford	1274370	115520	1389890	1274761	115895	1390656	0.06%
24	cnr-2000	540695	180711	721406	542278	182286	724564	0.44%
25	amazon0505	-	-	-	-	-	-	-
26	neos	30850	16008	46858	31082	16211	47293	0.92%
27	mc2depi	-	-	-	-	-	-	-
28	flickr	7812107	669854	8481961	7832994	690718	8523712	0.49%
29	web-Google	2050689	211510	2262199	2050427	211220	2261647	-0.02%
30	webbase-1M	489524	106337	595861	497840	114642	612482	2.71%

ANNEXE F TEMPS D'EXÉCUTION DE LA SPMV

Tableau F.1 : Temps d'exécution en millisecondes moyen mesuré pour chaque noyau sur le processeur GeForce GTX 670

#	Matrices	CSR		ELL	COO	HYB		
		Un fil par ligne	Une chaîne par ligne			Section ELL	Section COO	Total
1	Harvard500	6.71E-02	1.53E-02	1.77E-01	3.52E-02	1.57E-02	3.66E-02	5.23E-02
2	rail4284	3.63E+01	2.57E+00	6.14E+01	6.39E+00	3.95E+00	3.49E+00	7.44E+00
3	wiki-Vote	2.81E-01	6.81E-02	8.66E-01	9.63E-02	1.63E-02	8.95E-02	1.06E-01
4	California	4.81E-02	5.03E-02	1.61E-01	3.90E-02	1.37E-02	3.74E-02	5.10E-02
5	wb-cs-stanford	9.47E-02	5.31E-02	2.72E-01	5.13E-02	1.53E-02	4.16E-02	5.69E-02
6	Linverse	4.02E-02	6.92E-02	2.19E-02	7.78E-02	2.23E-02	-	2.23E-02
7	ex11	1.63E+00	1.52E-01	1.94E-01	6.56E-01	1.82E-01	2.99E-02	2.12E-01
8	OPF_6000	1.81E-01	1.61E-01	2.04E-01	1.87E-01	5.16E-02	6.90E-02	1.21E-01
9	pdb1HYS	6.45E+00	4.23E-01	6.69E-01	2.45E+00	5.14E-01	1.35E-01	6.49E-01
10	OPF_10000	2.76E-01	2.34E-01	2.19E-01	2.81E-01	7.17E-02	7.59E-02	1.48E-01
11	rma10	2.98E+00	3.57E-01	5.39E-01	1.34E+00	2.70E-01	2.70E-01	5.41E-01
12	conf5_4-8x8-05	2.45E+00	3.92E-01	1.75E-01	1.15E+00	1.74E-01	-	1.74E-01
13	nasasrb	3.64E+00	4.10E-01	7.96E-01	1.55E+00	2.69E-01	3.28E-02	3.01E-01
14	cant	5.80E+00	5.22E-01	4.53E-01	2.30E+00	4.42E-01	3.42E-02	4.76E-01
15	finan512	2.15E-01	3.98E-01	2.80E-01	3.80E-01	6.12E-02	1.10E-01	1.71E-01
16	consph	8.95E+00	7.62E-01	6.05E-01	3.45E+00	6.05E-01	-	6.05E-01
17	hcircuit	4.97E-01	5.33E-01	6.73E+00	3.43E-01	6.51E-02	7.30E-02	1.38E-01
18	cop20k_A	3.11E+00	7.59E-01	7.68E-01	1.64E+00	3.56E-01	3.01E-01	6.57E-01
19	shipsec1	3.34E+00	9.31E-01	7.73E-01	2.14E+00	4.38E-01	3.14E-01	7.51E-01
20	scircuit	3.53E-01	8.66E-01	2.90E+00	6.30E-01	1.15E-01	1.80E-01	2.95E-01
21	mac_econ_fwd500	5.13E-01	1.06E+00	6.72E-01	8.25E-01	1.85E-01	2.18E-01	4.02E-01
22	pwtk	1.62E+01	1.58E+00	2.30E+00	6.54E+00	1.00E+00	5.52E-02	1.06E+00
23	web-Stanford	1.99E+00	1.91E+00	4.94E+00	1.97E+00	5.46E-01	9.94E-01	1.54E+00
24	cnr-2000	2.79E+00	1.63E+00	-	1.95E+00	2.46E-01	1.17E+00	1.41E+00
25	amazon0505	2.60E+00	2.30E+00	1.47E+00	2.46E+00	1.47E+00	-	1.47E+00
26	neos	2.61E-01	2.34E+00	7.47E-01	9.99E-01	1.68E-01	9.93E-02	2.67E-01
27	mc2depi	3.53E-01	2.62E+00	2.01E-01	1.29E+00	2.01E-01	-	2.01E-01
28	flickr	1.01E+01	6.06E+00	-	7.20E+00	3.68E-01	6.55E+00	6.92E+00
29	web-Google	5.88E+00	5.61E+00	-	5.83E+00	2.87E+00	2.12E+00	4.99E+00
30	webbase-1M	2.51E+00	5.09E+00	-	2.08E+00	2.24E-01	8.07E-01	1.03E+00

Tableau F.2 : Temps d'exécution en millisecondes moyen mesuré pour chaque noyau sur le processeur Tesla K20c

#	Matrices	CSR		ELL	COO	HYB		
		Un fil par ligne	Une chaîne par ligne			Section ELL	Section COO	Total
1	Harvard500	1.00E-01	2.42E-02	2.36E-01	3.62E-02	2.86E-02	4.14E-02	7.00E-02
2	rail4284	4.20E+01	2.47E+00	9.13E+01	8.12E+00	4.76E+00	4.45E+00	9.21E+00
3	wiki-Vote	4.12E-01	6.66E-02	1.31E+00	1.27E-01	2.63E-02	1.19E-01	1.45E-01
4	California	8.21E-02	4.85E-02	2.54E-01	6.16E-02	2.24E-02	5.71E-02	7.95E-02
5	wb-cs-stanford	1.55E-01	5.44E-02	4.38E-01	7.67E-02	2.65E-02	6.37E-02	9.02E-02
6	Linverse	4.71E-02	5.81E-02	3.12E-02	1.07E-01	3.17E-02	-	3.17E-02
7	ex11	9.02E-01	1.41E-01	1.73E-01	8.51E-01	1.59E-01	4.32E-02	2.02E-01
8	OPF_6000	1.61E-01	1.36E-01	2.27E-01	2.44E-01	5.95E-02	9.77E-02	1.57E-01
9	pdb1HYS	4.37E+00	3.77E-01	6.92E-01	3.17E+00	5.36E-01	1.79E-01	7.15E-01
10	OPF_10000	2.29E-01	1.95E-01	2.15E-01	3.71E-01	7.82E-02	1.03E-01	1.81E-01
11	rma10	1.92E+00	3.10E-01	5.12E-01	1.74E+00	2.49E-01	3.50E-01	6.00E-01
12	conf5_4-8x8-05	1.48E+00	3.53E-01	1.73E-01	1.49E+00	1.72E-01	-	1.72E-01
13	nasasrb	2.17E+00	3.59E-01	9.17E-01	2.01E+00	3.17E-01	4.96E-02	3.67E-01
14	cant	3.83E+00	4.59E-01	4.47E-01	2.99E+00	4.37E-01	5.07E-02	4.88E-01
15	finan512	1.96E-01	3.33E-01	2.69E-01	4.89E-01	6.44E-02	1.43E-01	2.08E-01
16	consph	6.24E+00	6.69E-01	6.09E-01	4.47E+00	6.08E-01	-	6.08E-01
17	hcircuit	7.80E-01	4.26E-01	5.87E+00	4.42E-01	6.76E-02	1.00E-01	1.68E-01
18	cop20k_A	1.64E+00	6.77E-01	6.78E-01	2.12E+00	3.27E-01	3.94E-01	7.21E-01
19	shipsec1	1.95E+00	8.02E-01	7.12E-01	2.78E+00	4.05E-01	3.90E-01	7.96E-01
20	scircuit	3.75E-01	6.88E-01	2.59E+00	8.13E-01	1.09E-01	2.27E-01	3.36E-01
21	mac_econ_fwd500	4.46E-01	8.39E-01	6.18E-01	1.07E+00	1.74E-01	2.75E-01	4.49E-01
22	pwtk	1.22E+01	1.36E+00	2.12E+00	8.42E+00	9.35E-01	7.80E-02	1.01E+00
23	web-Stanford	1.40E+00	1.56E+00	4.15E+00	2.27E+00	3.62E-01	1.15E+00	1.51E+00
24	cnr-2000	2.34E+00	1.38E+00	-	2.51E+00	2.37E-01	1.50E+00	1.73E+00
25	amazon0505	1.34E+00	1.81E+00	8.27E-01	2.92E+00	8.27E-01	-	8.27E-01
26	neos	2.50E-01	1.83E+00	6.86E-01	1.28E+00	1.71E-01	1.31E-01	3.02E-01
27	mc2depi	3.24E-01	2.04E+00	2.08E-01	1.67E+00	2.08E-01	-	2.08E-01
28	flickr	8.27E+00	5.01E+00	-	8.71E+00	2.73E-01	7.95E+00	8.22E+00
29	web-Google	3.75E+00	4.43E+00	1.96E+01	5.77E+00	1.90E+00	2.00E+00	3.91E+00
30	webbase-1M	3.16E+00	3.87E+00	-	2.52E+00	2.33E-01	9.70E-01	1.20E+00

ANNEXE G TEMPS D'EXÉCUTION ESTIMÉ DE LA SPMV

Tableau G.1 : Temps d'exécution estimé en millisecondes pour chaque implémentation sur le processeur GeForce GTX 670

#	Matrices	CSR		ELL	COO	HYB
		Un fil par ligne	Une chaîne par ligne			
1	Harvard500	2.47E-03	2.46E-03	4.77E-03	7.77E-04	7.01E-04
2	rail4284	2.25E+01	2.50E+00	1.83E+01	3.00E+00	5.66E+00
3	wiki-Vote	1.86E-01	7.89E-02	3.96E-01	5.99E-02	5.95E-02
4	California	1.69E-02	3.47E-02	4.27E-02	8.12E-03	7.94E-03
5	wb-cs-stanford	3.31E-02	4.09E-02	1.26E-01	1.08E-02	1.06E-02
6	Linverse	4.54E-02	5.37E-02	1.32E-02	1.91E-02	1.32E-02
7	ex11	1.54E+00	2.52E-01	1.88E-01	2.81E-01	1.83E-01
8	OPF_6000	2.37E-01	1.76E-01	1.68E-01	8.09E-02	8.44E-02
9	pdb1HYS	6.23E+00	7.39E-01	9.49E-01	8.95E-01	8.49E-01
10	OPF_10000	3.78E-01	2.62E-01	2.23E-01	1.24E-01	1.32E-01
11	rma10	3.27E+00	5.03E-01	6.65E-01	5.54E-01	4.91E-01
12	conf5_4-8x8-05	2.65E+00	6.00E-01	1.77E-01	6.03E-01	1.77E-01
13	nasasrb	3.72E+00	5.96E-01	9.03E-01	6.62E-01	4.12E-01
14	cant	5.63E+00	9.02E-01	6.68E-01	1.00E+00	6.63E-01
15	finan512	4.08E-01	4.88E-01	1.90E-01	1.81E-01	9.81E-02
16	consph	8.40E+00	1.41E+00	9.27E-01	1.58E+00	9.27E-01
17	hcircuit	2.51E-01	5.37E-01	5.86E+00	1.36E-01	9.33E-02
18	cop20k_A	3.55E+00	1.21E+00	1.20E+00	1.04E+00	8.42E-01
19	shipsec1	4.64E+00	1.41E+00	1.24E+00	1.18E+00	9.72E-01
20	scircuit	6.19E-01	9.28E-01	2.70E+00	3.17E-01	2.54E-01
21	mac_econ_fwd500	1.05E+00	1.21E+00	8.37E-01	4.76E-01	4.39E-01
22	pwtk	1.59E+01	2.32E+00	2.59E+00	2.65E+00	1.49E+00
23	web-Stanford	3.38E+00	2.58E+00	4.78E+00	1.89E+00	1.78E+00
24	cnr-2000	3.71E+00	1.69E+00	-	8.78E-01	8.05E-01
25	amazon0505	3.20E+00	3.00E+00	1.97E+00	1.81E+00	1.97E+00
26	neos	4.48E-01	2.58E+00	7.67E-01	5.25E-01	2.58E-01
27	mc2depi	5.39E-01	2.88E+00	2.33E-01	5.54E-01	2.33E-01
28	flickr	1.79E+01	7.99E+00	-	6.14E+00	6.10E+00
29	web-Google	6.91E+00	6.40E+00	-	4.32E+00	4.26E+00
30	webbase-1M	1.94E+00	5.04E+00	-	9.10E-01	6.45E-01

Tableau G.2 : Différence absolue entre le temps d'exécution estimé et le temps moyen mesuré pour chaque implémentation sur le processeur GeForce GTX 670

#	Matrices	CSR		ELL	COO	HYB
		Un fil par ligne	Une chaîne par ligne			
1	Harvard500	96.32%	83.88%	97.30%	97.79%	95.54%
2	rail4284	38.10%	2.73%	70.15%	53.02%	43.41%
3	wiki-Vote	33.64%	15.95%	54.26%	37.79%	264.58%
4	California	64.77%	31.02%	73.56%	79.19%	41.86%
5	wb-cs-stanford	65.05%	22.97%	53.57%	78.87%	30.83%
6	Linverse	13.03%	22.39%	39.49%	75.38%	40.73%
7	ex11	5.79%	65.49%	2.99%	57.21%	0.28%
8	OPF_6000	31.10%	9.10%	17.82%	56.82%	63.61%
9	pdb1HYS	3.44%	74.86%	41.98%	63.46%	65.06%
10	OPF_10000	37.27%	11.72%	2.01%	56.01%	84.56%
11	rma10	9.78%	43.33%	23.42%	58.73%	81.73%
12	conf5_4-8x8-05	8.17%	52.99%	1.21%	47.41%	1.66%
13	nasasrb	2.25%	45.23%	13.43%	57.30%	53.28%
14	cant	3.00%	72.78%	47.45%	56.60%	49.84%
15	finan512	89.51%	22.62%	31.98%	52.38%	60.22%
16	consph	6.11%	78.59%	53.25%	54.28%	53.23%
17	hcircuit	49.48%	0.75%	13.00%	60.29%	43.44%
18	cop20k_A	14.12%	60.33%	55.85%	36.76%	136.98%
19	shipsec1	38.83%	52.24%	60.96%	44.74%	122.17%
20	scircuit	75.30%	7.23%	6.77%	49.74%	120.60%
21	mac_econ_fwd500	104.13%	14.42%	24.63%	42.34%	137.51%
22	pwtk	1.87%	46.56%	12.75%	59.43%	48.24%
23	web-Stanford	69.76%	35.29%	3.10%	3.79%	225.39%
24	cnr-2000	33.01%	3.53%	-	55.00%	227.17%
25	amazon0505	22.77%	30.66%	34.13%	26.59%	34.09%
26	neos	71.50%	10.43%	2.64%	47.48%	54.05%
27	mc2depi	52.66%	9.74%	15.87%	56.95%	15.64%
28	flickr	76.95%	32.01%	-	14.77%	1557.35%
29	web-Google	17.61%	14.07%	-	26.04%	48.34%
30	webbase-1M	22.95%	1.08%	-	56.25%	187.97%

Tableau G.3 : Temps d'exécution en millisecondes estimé pour chaque implémentation sur le processeur Tesla K20c

#	Matrices	CSR		ELL	COO	HYB
		Un fil par ligne	Une chaîne par ligne			
1	Harvard500	2.28E-03	2.28E-03	4.40E-03	7.18E-04	6.48E-04
2	rail4284	2.08E+01	2.31E+00	1.69E+01	2.77E+00	5.23E+00
3	wiki-Vote	1.72E-01	7.29E-02	3.66E-01	5.53E-02	5.50E-02
4	California	1.57E-02	3.21E-02	3.94E-02	7.51E-03	7.34E-03
5	wb-cs-stanford	3.06E-02	3.78E-02	1.17E-01	1.00E-02	9.81E-03
6	Linverse	4.20E-02	4.96E-02	1.22E-02	1.77E-02	1.22E-02
7	ex11	1.42E+00	2.33E-01	1.74E-01	2.60E-01	1.69E-01
8	OPF_6000	2.19E-01	1.62E-01	1.55E-01	7.48E-02	7.80E-02
9	pdb1HYS	5.75E+00	6.83E-01	8.77E-01	8.27E-01	7.84E-01
10	OPF_10000	3.49E-01	2.42E-01	2.06E-01	1.14E-01	1.22E-01
11	rma10	3.02E+00	4.65E-01	6.14E-01	5.12E-01	4.54E-01
12	conf5_4-8x8-05	2.45E+00	5.55E-01	1.63E-01	5.57E-01	1.63E-01
13	nasasrb	3.44E+00	5.51E-01	8.34E-01	6.12E-01	3.80E-01
14	cant	5.20E+00	8.34E-01	6.18E-01	9.24E-01	6.12E-01
15	finan512	3.77E-01	4.51E-01	1.76E-01	1.67E-01	9.07E-02
16	consph	7.77E+00	1.30E+00	8.57E-01	1.46E+00	8.57E-01
17	hcircuit	2.32E-01	4.96E-01	5.41E+00	1.26E-01	8.62E-02
18	cop20k_A	3.28E+00	1.12E+00	1.11E+00	9.60E-01	7.78E-01
19	shipsec1	4.29E+00	1.30E+00	1.15E+00	1.09E+00	8.98E-01
20	scircuit	5.72E-01	8.58E-01	2.50E+00	2.93E-01	2.35E-01
21	mac_econ_fwd500	9.68E-01	1.12E+00	7.73E-01	4.40E-01	4.05E-01
22	pwtk	1.47E+01	2.14E+00	2.40E+00	2.45E+00	1.38E+00
23	web-Stanford	3.12E+00	2.38E+00	4.42E+00	1.75E+00	1.64E+00
24	cnr-2000	3.43E+00	1.56E+00	-	8.11E-01	7.44E-01
25	amazon0505	2.95E+00	2.77E+00	1.82E+00	1.67E+00	1.82E+00
26	neos	4.14E-01	2.39E+00	7.08E-01	4.85E-01	2.39E-01
27	mc2depi	4.98E-01	2.66E+00	2.15E-01	5.12E-01	2.15E-01
28	flickr	1.65E+01	7.39E+00	-	5.67E+00	5.64E+00
29	web-Google	6.38E+00	5.92E+00	1.88E+01	3.99E+00	3.93E+00
30	webbase-1M	1.79E+00	4.65E+00	-	8.41E-01	5.96E-01

Tableau G.4 : Différence absolue entre le temps d'exécution estimé et le temps moyen mesuré pour chaque implémentation sur le processeur Tesla K20c

#	Matrices	CSR		ELL	COO	HYB
		Un fil par ligne	Une chaîne par ligne			
1	Harvard500	97.72%	90.59%	98.14%	98.02%	97.73%
2	rail4284	50.62%	6.25%	81.44%	65.82%	9.88%
3	wiki-Vote	58.22%	9.55%	72.15%	56.40%	108.60%
4	California	80.93%	33.88%	84.47%	87.81%	67.28%
5	wb-cs-stanford	80.24%	30.55%	73.34%	86.94%	63.04%
6	Linverse	10.94%	14.63%	60.79%	83.40%	61.40%
7	ex11	57.63%	65.12%	0.36%	69.51%	6.19%
8	OPF_6000	36.49%	19.38%	31.88%	69.41%	31.15%
9	pdb1HYS	31.59%	81.37%	26.69%	73.90%	46.32%
10	OPF_10000	52.71%	24.05%	4.01%	69.17%	56.39%
11	rma10	57.47%	52.54%	20.05%	70.64%	82.26%
12	conf5_4-8x8-05	65.95%	57.38%	5.60%	62.68%	5.18%
13	nasasrb	58.44%	53.30%	9.06%	69.59%	20.01%
14	cant	35.86%	81.57%	38.09%	69.14%	40.18%
15	finan512	92.16%	35.34%	34.57%	65.79%	40.70%
16	consph	24.40%	87.98%	40.74%	67.46%	40.85%
17	hcircuit	70.25%	16.48%	7.80%	71.47%	27.58%
18	cop20k_A	99.51%	66.05%	63.02%	54.68%	137.79%
19	shipsec1	119.46%	63.28%	61.55%	60.70%	121.76%
20	scircuit	52.81%	24.72%	3.61%	64.01%	115.83%
21	mac_econ_fwd500	117.11%	33.45%	25.17%	58.85%	133.25%
22	pwtk	20.48%	57.89%	13.14%	70.88%	47.12%
23	web-Stanford	122.15%	52.46%	6.48%	22.88%	353.71%
24	cnr-2000	46.68%	13.52%	-	67.68%	213.41%
25	amazon0505	120.52%	52.98%	120.32%	42.89%	120.24%
26	neos	65.70%	30.26%	3.33%	62.26%	39.51%
27	mc2depi	53.47%	30.63%	3.22%	69.39%	3.33%
28	flickr	99.83%	47.47%	-	34.90%	1961.72%
29	web-Google	70.15%	33.57%	3.93%	30.94%	106.48%
30	webbase-1M	43.36%	20.25%	-	66.70%	155.31%